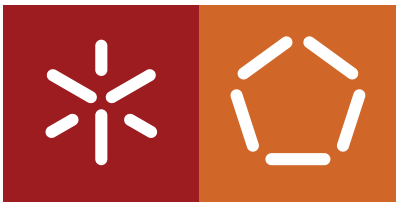
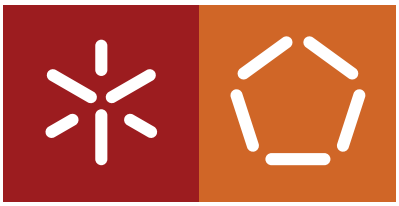


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Diogo da Silva Sá

Aplicações Java em Arquiteturas Paralelas de Acesso Não Uniforme à Memória

12 de Dezembro de 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Diogo da Silva Sá

Aplicações Java em Arquiteturas Paralelas de Acesso Não Uniforme à Memória

Dissertação de Mestrado
Obtenção do Grau de Mestre em Engenharia Informática

Dissertação sob a orientação de:
Professor João Luís Ferreira Sobral
Engenheiro Bruno Silvestre Medeiros

12 de Dezembro de 2017

ABSTRACT

During several decades, computers performance improvement was mostly achieved by increasing the clock frequency of the processor. However, increasing the clock frequency became technically harder due to problems of dissipation and energy consumption. To deal with this problems, computer architectures started to go in the direction of increasing the number of processors. The memory architectures initially adopted namely *Uniform Memory Access (UMA)*, presented scalability limitations. The Symmetric Multiprocessing (SMP) has multiple processors (potentially multicore) that access main memory uniformly using a system bus shared among all processors. This led to contention when accessing main memory.

The *Non-Uniform Memory Access (NUMA)* came up during the 90's with multiple memory banks. In NUMA systems, the number of memory banks can increase with the number of processors also scaling the overall memory bandwidth. These and other characteristics become especially attractive to speedup a class of algorithms whose scalability is limited by main memory performance designated by *memory-bound* algorithms. However, programmers who want to take full advantage of this memory architecture have to deal with new challenges, namely thread and memory affinity.

This dissertation shows that how *memory-bound* algorithms access memory in NUMA architectures can have a significant impact on performance. A set of tests were used to identify how different techniques and affinity strategies can help improving the performance of Java and C algorithms in NUMA architectures. The tests were performed using different approaches as operating system tools, JVM options, programming techniques or affinity variables for the compilers. These approaches allowed to increase the scalability of two case studies in a NUMA architecture giving an additional speedup for Java applications up to 1.8 times, and up to 2.16 times for C implementations.

RESUMO

Durante várias décadas, o aumento do desempenho dos processadores era conseguido maioritariamente através do aumento da frequência de relógio. Contudo, aumentar o desempenho através do aumento da frequência tornou-se cada vez mais difícil conduzindo a problemas de consumo energético e dissipação de calor. Para resolver estes problemas, as arquiteturas mais recentes evoluíram num sentido de aumentar o número de processadores, e mais tarde, o número de núcleos. As arquiteturas de memória inicialmente adotadas, designadas arquiteturas de acesso uniforme à memória (UMA), apresentaram algumas limitações. Entre as arquiteturas UMA existentes, existe a arquitetura de multiprocessamento simétrico (SMP). Esta arquitetura possui múltiplos processadores que acedem à memória principal utilizando um barramento partilhado pelos processadores que conduziu a problemas de contenção no acesso à memória principal.

As arquiteturas de acesso não uniforme à memória (NUMA) surgiram durante os anos 90 com múltiplos bancos de memória. No entanto, os programadores que queiram tirar total partido das vantagens desta arquitetura, terão que lidar com novos desafios ao nível da programação, ao nível da afinidade dos fios de execução e das alocações de memória.

Esta dissertação mostra que a forma como os algoritmos *memory-bound* acedem a dados de memória principal numa arquitetura NUMA, pode ter impacto no seu desempenho. Um conjunto de testes foi utilizado para demonstrar em que medida várias técnicas de afinidade podem contribuir para aumentar o desempenho de aplicações Java e C em arquiteturas NUMA. Os testes realizados com recurso a diferentes abordagens tais como ferramentas do sistema operativo, opções da JVM, técnicas de programação e até variáveis de afinidade para os compiladores, foram usados para aumentar a desempenho de dois casos de estudo em arquiteturas NUMA. A utilização destas abordagens permitiram aumentar o desempenho com um ganho adicional de até 1.8 vezes para as implementações em Java, e de até 2.16 vezes para as versões em C das mesmas aplicações.

AGRADECIMENTOS

A realização da presente dissertação contou com o apoio e incentivo de algumas pessoas sem as quais não se teria tornado uma realidade e a quem estarei eternamente grato.

Ao Professor João Luís Sobral, pela orientação e dedicação dadas a esta dissertação. O seu acompanhamento e críticas construtivas ao longo das reuniões semanais foram fundamentais na criação deste trabalho. Ao meu coorientador, Bruno Silvestre Medeiros, pelo seu papel ativo e construtivo ao longo de todo o projeto com sugestões e colaboração no trabalho de revisão. Um agradecimento adicional ao Professor Doutor António Pina pelo interesse e apoio demonstrados. À minha família, a quem tudo devo por me terem proporcionado o meu percurso académico. Em especial, à minha mãe, Emília Silva, por tudo o que fez por mim ao longo destes anos. Um enorme e sincero obrigado pelo suporte que jamais serei capaz de retribuir e obrigado pela minha educação ter contado com a mãe mais bondosa deste mundo. Aos meus colegas do laboratório de Arquitetura de Computadores, pela companhia e apoio em sessões de trabalho muitas vezes até horas tardias. A todos os meus amigos um sincero obrigado pelo apoio emocional e por me incentivarem constantemente a abraçar novos desafios. Por fim e não menos importante, um forte agradecimento à minha namorada, Fernanda Névoa, pelo apoio incondicional.

CONTEÚDO

LISTA DE FIGURAS

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Nas últimas décadas, o desempenho dos microprocessadores aumentou a uma taxa consideravelmente superior comparativamente ao aumento do desempenho da memória. O aumento deste hiato entre processador e memória (conhecido como *"The Memory Gap"*) tornou-se um dos principais motivos para que fosse cada vez mais difícil aumentar o desempenho dos sistemas de computação [25][43]. Assim, ao longo das últimas décadas, aumentou a preocupação de propor arquiteturas que aumentassem a velocidade de acesso à memória ao invés de utilizar processadores cada vez mais rápidos [43].

Os sistemas de multiprocessador com arquiteturas de acesso uniforme à memória (UMA) são dotados de vários processadores que partilham o mesmo banco de memória principal. Se os processadores nesta arquitetura utilizarem um barramento de memória, esta arquitetura é alternativamente designada na literatura por arquitetura de multiprocessamento simétrico (*Symmetric Multiprocessing - SMP*).

Este tipo de arquitetura foi implementada no início dos anos 60 numa máquina *Burroughs D8250*[42, 44]. O desenho deste tipo de arquitetura torna a organização dos sistemas multiprocessador simples e flexível. Esse desenho permite uma fácil expansão por forma a aumentar o desempenho de forma incremental adicionando mais processadores ao barramento de memória[37]. Contudo, sendo este um barramento de memória partilhado pelos processadores, o aumento do número de processadores resulta num aumento da contenção no acesso à memória principal. Esta contenção contribui para que o desenho destas arquiteturas acabe por limitar a escalabilidade do sistema tornando o acesso à memória gradualmente mais lento[17].

A existência de múltiplos núcleos de processamento e a consequente introdução de diferentes níveis de *cache*, pode aliviar o problema de contenção mas obriga ao uso de protocolos para prevenir problemas de coerência de *cache* (e.g. protocolo MESI) gerando tráfego adicional no barramento.

Nos anos 90 surgiu uma nova arquitetura para sistemas multiprocessador com acesso não uniforme à memória (*Non-uniform memory access - NUMA*). Com esta arquitetura surge uma nova abordagem em relação à organização da memória com o objetivo de aumentar a velocidade de acesso à memória principal ultrapassando limitações de escalabilidade existentes nas arquiteturas SMP. Nas arquiteturas NUMA, em vez de haver um único barramento partilhado pelos processadores existe um banco de memória ligado diretamente a cada um deles[37]. Cada processador comunica com outros processadores através de barramentos de elevada largura de banda e de baixa latência, proporcionando maior escalabilidade que os barramentos de memória partilhados[37]. Os acessos à memória ocorrem de forma não uniforme sendo que um processador pode aceder a dados que estão no banco de memória que lhe está diretamente ligado (designada de memória local) com uma latência inferior à necessária para aceder ao banco de memória de outro processador (memória remota).

As características das arquiteturas NUMA são especialmente atrativas para reduzir os custos com acessos à memória em algoritmos *memory-bound*. No entanto, para se tirar o máximo partido deste tipo de arquiteturas é necessário um esforço adicional ao nível da distribuição dos fios de execução em aplicações paralelas, e à forma como as aplicações alocam e acedem a dados na memória[45].

A linguagem Java é uma linguagem portável e segura, sendo uma linguagem muito utilizada atualmente. Dado o seu suporte nativo para aplicações paralelas, têm crescido as preocupações relacionadas com o desempenho. Em arquiteturas NUMA, o aumento de desempenho das aplicações está dependente do modo como os processadores acedem à memória. O controlo sobre a afinidade dos fios de execução e sobre as alocações de memória pelos diferentes nós NUMA em Java é, no entanto, limitado quando comparado com outras linguagens como a linguagem C. As otimizações em Java para arquiteturas NUMA estão pré-definidas na JVM não sendo controláveis pelo programador. Existem outras abordagens que aumentam o nível de suporte mas que requerem a utilização de ferramentas externas ou bibliotecas resultando num maior esforço de programação. Os

testes realizados na dissertação irão estudar este tipo de otimizações com o objetivo de aumentar o desempenho das aplicações em Java para este tipo de arquiteturas.

1.2 OBJETIVOS

As arquiteturas NUMA surgiram como uma abordagem alternativa às arquiteturas SMP aumentando o desempenho dos acessos à memória. Esta dissertação foca-se no estudo de mecanismos ao nível da programação e da execução de programas paralelos que permitam aumentar o seu desempenho em arquiteturas NUMA. Com esta finalidade, a dissertação decompõe-se em três objetivos fundamentais:

- Analisar o impacto das arquiteturas NUMA em aplicações paralelas escritas em Java;
- Estudar técnicas que permitam aumentar o desempenho das aplicações paralelas neste tipo de arquiteturas;
- Identificar as principais limitações específicas do Java na otimização de aplicações para arquiteturas NUMA.

Pretende-se com este estudo avaliar o nível de suporte do Java para as arquiteturas NUMA, detetando possíveis limitações ou entraves. O estudo é validado com dois casos de estudo implementados em Java comparando os resultados de desempenho com implementações na linguagem *C*.

1.3 ESTRUTURA DA DISSERTAÇÃO

O desenvolvimento desta tese começou pelo estudo das arquiteturas multiprocessador onde se incluem as arquiteturas SMP e as arquiteturas NUMA. O capítulo 2 apresenta o estado da arte com introdução de arquiteturas paralelas de memória partilhada onde se incluem as arquiteturas SMP e NUMA. Estas são apresentadas detalhadamente quanto à organização de memória identificando as principais limitações que estas impõe à escalabilidade das aplicações.

Numa fase mais avançada foram explorados mecanismos e ferramentas de controlo de afinidade de fios de execução e de dados. Foram igualmente exploradas várias estratégias de afinidade que estes mecanismos e ferramentas possibilitam, como um meio de aumentar o desempenho de aplicações em arquiteturas NUMA. Nesta fase incluem-se

ainda algumas técnicas de otimização de desempenho ao nível da programação e o estudo do nível de suporte atual do Java para este tipo de arquiteturas. O trabalho relativo a esta fase é apresentado no capítulo 3.

O capítulo 4 apresenta o trabalho experimental desenvolvido no decorrer desta dissertação onde se testaram os mecanismos de controlo de afinidade e técnicas de programação identificados. Este capítulo começa por fazer uma apresentação dos casos de estudo e das suas implementações em Java. Em seguida é apresentado o estudo realizado do desempenho dessas aplicações com testes que exploram diferentes configurações de afinidade, e técnicas de programação com vista à melhoria do desempenho num nó computacional NUMA com 2 processadores com um total de 32 núcleos.

O capítulo 5 apresenta uma sistematização dos resultados obtidos acompanhados de uma reflexão crítica e sugestões em relação a novos desafios para trabalho futuro.

2

ARQUITETURAS PARALELAS DE MEMÓRIA PARTILHADA

2.1 INTRODUÇÃO

Ao longo das últimas décadas foram surgindo arquiteturas de computadores com diferentes abordagens relativamente ao desenho da memória. Este desenho foi sofrendo alterações introduzindo diferentes formas de organizar o espaço de endereçamento e transporte de dados entre processador e a memória. Estas alterações surgiram à medida que as arquiteturas multiprocessador assumiam um papel cada vez mais importante na construção de servidores e supercomputadores, conseguindo atingir um maior desempenho comparativamente às arquiteturas com um único processador (também designadas de uniprocessador). As arquiteturas uniprocessador verificaram maiores dificuldades no aumento do desempenho que decorriam das limitações de exploração de mais paralelismo ao nível das instruções (ILP). O aumento do desempenho em arquiteturas multiprocessador foi conseguido seguindo uma lógica de replicação de recursos explorando o paralelismo ao nível do fio de execução (*thread-level paralelism ou TLP*)[20].

As arquiteturas multiprocessador consistiam inicialmente num conjunto de processadores acoplados, cujo o controlo está a cargo de um único sistema operativo. Os processadores nesta arquitetura utilizam um único espaço de endereçamento partilhado por todos os processadores e por essa razão estas arquiteturas são usualmente designadas por arquiteturas de memória partilhada. A utilização de um espaço de endereçamento único e partilhado por diferentes processadores faz com que todos os processadores possuam a mesma visão sobre os dados. Isto não significa, no entanto, que apenas exista um banco de memória física[20]. A figura 1 ilustra o modelo típico da arquitetura multiprocessador de memória partilhada com 16 processadores (imagem (a) da esquerda) e

uma ilustração do espaço de endereçamento particionado em 16 secções distintas cada uma tratada por um processador diferente[39] (imagem (b) à direita)[1].

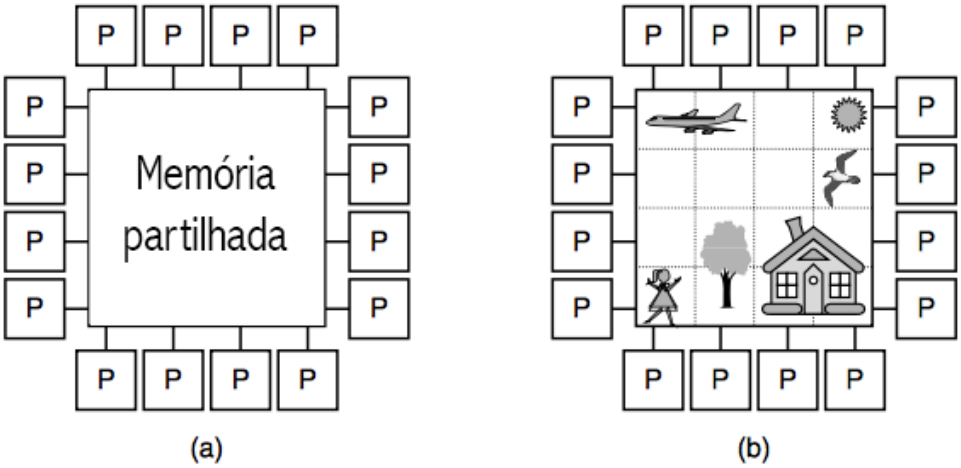


Figura 1.: Ilustração do modelo típico de uma arquitetura multiprocessador de memória partilhada.

Nas últimas décadas surgiram processadores com múltiplos núcleos de processamento no mesmo processador. Estes processadores suportam a execução de múltiplos fios de execução em paralelo aumentando o desempenho sem aumentar a frequência interna do relógio.

As memórias *cache* são memórias de rápido acesso que possuem um custo por byte armazenado consideravelmente mais elevado que as memórias DRAM. Equipar cada processador com memórias *cache* de maior tamanho faz com que seja possível diminuir a latência média de acesso aos dados mantendo a cópia de um maior volume de dados em memória cache[20]. As memórias *cache* desempenham um papel importante em arquiteturas de memória partilhada pois reduzem o tempo de acesso aos dados e diminuem a contenção no acesso à memória partilhada.

A próxima secção explora uma variante das arquiteturas UMA, designada por multiprocessamento simétrico (*Symmetric Multiprocessing* ou SMP). Esta arquitetura é formada por um conjunto de processadores independentes de desempenho semelhante que

1 Imagem de *Structured Computer Organization 5ºEdição (2005) por Andrew Tanenbau (adaptada)*

utilizam um barramento partilhado para transferência de dados entre processadores e memória [37]. De seguida, será explorada uma arquitetura de memória partilhada alternativa às arquiteturas UMA igualmente assente numa lógica MIMD[13] de acordo com a taxonomia introduzida por Flynn no inicio dos anos 70[13]. A arquitetura NUMA (Non-uniform Memory Access - NUMA) surgiu depois das arquiteturas SMP e será a arquitetura onde todo o trabalho experimental desta dissertação assenta. Estes dois tipos de arquiteturas serão exploradas nas próximas secções em mais detalhe.

2.2 MULTIPROCESSAMENTO SIMÉTRICO

2.2.1 *Visão geral*

As arquiteturas de multiprocessamento simétrico (SMP) são uma variante das arquiteturas UMA em que um conjunto de processadores independentes de desempenho semelhante[37] partilham a mesma memória principal através de um barramento partilhado. Sobre este barramento são realizadas as tarefas de endereçamento e transporte de dados entre processadores e memória. Nesta arquitetura, à semelhança de outras variantes das arquiteturas UMA, cada processador tem um tempo de acesso à memória física uniforme.

Na figura 2 está representada uma imagem clássica da arquitetura SMP :

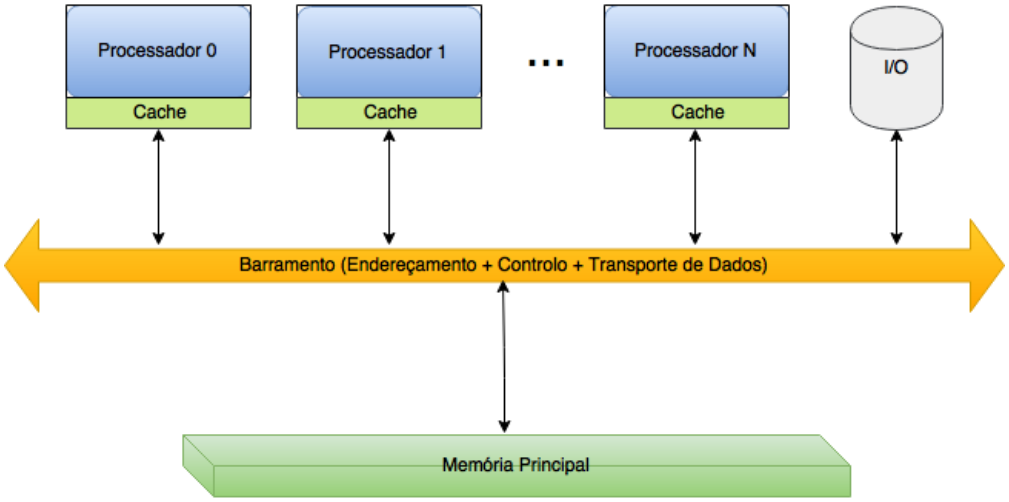


Figura 2.: Sistema de multiprocessamento simétrico

Nas arquiteturas SMP todos os fios de execução em múltiplos núcleos de processamento partilham um único espaço de endereçamento lógico. Os fios de execução leem e escrevem dados realizando operações de carregamento (*load*) e armazenamento (*store*) através da memória partilhada[37].

Os processadores a azul na figura 2 comunicam com o barramento (em amarelo na figura 2) utilizando a memória[20]. O sistema operativo faz com que a existência de múl-

tiplos processadores seja transparente ao utilizador e encarrega-se das tarefas de escalonamento de tarefas, comunicação e sincronização entre diferentes processadores[37].

Quando a arquitetura SMP é equipada com processadores multinúcleo, o sistema operativo trata todos os núcleos como se de processadores independentes se tratassem. Esta característica extendeu-se ainda aos processadores multinúcleo com suporte para *Simultaneous multithreading (SMT)* como a tecnologia *Hyper-Threading*. Num processador com *HyperThreading*, cada núcleo de processamento é um núcleo físico mas apresenta-se ao sistema operativo como sendo dois ou mais processadores lógicos. Assim, a introdução de processadores multinúcleo nas arquiteturas SMP aumentou de forma considerável o número de processadores nesta arquitetura a aceder ao barramento de memória.

2.2.2 *Limitações das arquiteturas SMP*

O crescimento do número de processadores ao longo dos anos na arquitetura SMP teve como principal consequência o aumento do tráfego no barramento partilhado. Este aumento foi ainda mais agravado com a evolução da arquitetura para processadores multinúcleo. O aumento do número de núcleos fez com que se aumentasse o número de *processadores* vistos pelo sistema operativo. Dada a simplicidade do desenho, a arquitetura seguiu uma lógica incremental em que se adicionavam mais processadores ao barramento partilhado como forma de aumentar o desempenho. No entanto, os barramentos de acesso à memória não evoluíram na mesma medida ao longo dos anos por forma a acompanhar o aumento do tráfego. Assim, existe um limite nessa escalabilidade (geralmente entre 16 e 64 processadores) a partir do qual o aumento do desempenho não é garantido com a adição de mais processadores ao barramento[37].

Quando demasiados processadores acedem ao barramento, alguns desses processadores terão que esperar até que o barramento seja libertado[37]. Este aspecto representa um *bottleneck*[2] na arquitetura que pode ser minimizado com a utilização de memórias *cache* e mecanismos de caching por forma a diminuir o tráfego no barramento[20].

2 Um *bottleneck* é um termo utilizado em engenharia quando o desempenho de um dado sistema é limitado por um componente ou recurso deste.

A utilização de mecanismos de *caching* introduzem a necessidade de garantir a coerência de *cache* em todos os processadores[20] e com isso uma nova limitação na escalabilidade nas arquiteturas SMP.

2.3 ARQUITETURA DE ACESSO NÃO UNIFORME À MEMÓRIA (NUMA)

2.3.1 *Terminologia*

Em termos de organização de memória e barramentos, a arquitetura NUMA traz uma abordagem distinta de organizar a memória e a forma como os processadores realizam o acesso à mesma. Com o surgimento desta nova arquitetura foram introduzidos novos conceitos para referir diferentes partes da arquitetura e novos termos acerca do funcionamento da mesma. No entanto, estes conceitos variam na literatura pelo que é comum existirem termos distintos com o mesmo significado. Desta forma, torna-se necessário fixar alguma terminologia relevante para toda a dissertação.

- **NUMA**: É um acrónimo para **N**on-**U**niform **M**emory **A**ccess (acesso não uniforme à memória). Enquanto que nas arquiteturas de acesso uniforme à memória os diferentes processadores têm um mesmo tempo de acesso à memória principal, nas arquiteturas NUMA esse tempo varia em função do banco de memória que é acedido[15, 18].

- **Nó NUMA** (ou apenas **nó**): Refere-se ao banco de memória principal (DRAM) que está ligada fisicamente a um processador[18]. Esta memória é designada de *memória* ou *nó* **local** ao processador ao qual está ligado diretamente. Por outro lado, um processador pode aceder à memória de outro processador através de um conetor que os interligam. Essa memória é considerada memória **remota** e possui um tempo de acesso superior comparativamente ao acesso à memória local por causa da necessidade de travessia de um ou vários conetores que interligam os processadores[18].

- **Conetor (*interconnect*)**: Uma arquitetura NUMA consiste em múltiplos bancos de memória (nós) cada um destes ligado a um processador através de um barramento de memória. Esses processadores por sua vez são interligados entre si

através de conetores construidos com tecnologia que permite uma elevada largura de banda em comparação com o barramento de memória usado na arquitetura SMP. Os conectores na arquitetura NUMA tornam possível a comunicação entre processadores e fazem com que estes possam aceder à memória principal uns dos outros. Por vezes os conectores na literatura são também designados de *NUMA-links*[21].

- **Processador**: O termo *processador* será usado para referir um único *chip* que pode conter internamente múltiplos núcleos de processamento e que está localizado fisicamente num *socket*. Assim, os termos *processador*, *socket*, ou *processador multinúcleo* são referidos com o mesmo significado.

- **Núcleo**: Refere-se a uma unidade de processamento contida num processador. Um processador pode conter um ou múltiplos núcleos de processamento.

- **NUMA aware**: Refere-se à criação de algoritmos em linguagens de programação que são implementados tendo em consideração a organização de memória por forma a tirar partido das arquiteturas NUMA[22].

- **Hops**: É uma métrica que contabiliza o número de conetores (QPI, UPI, ou HyperTransport) envolvidos num acesso de memória para transporte dos dados do nó até ao processador que realiza o acesso. Quantos mais *hops* envolvidos no fluxo de dados, maior é a latência associada ao acesso remoto e menor é a largura de banda atingida[45].

2.3.2 *Organização de memória*

A arquitetura NUMA sugere uma forma alternativa de disponibilizar a memória aos processadores por forma a ultrapassar as limitações de escalabilidade existentes nas arquiteturas UMA. Nas arquiteturas NUMA, cada processador possui um banco de memória ligado a si diretamente por um barramento de memória. O tempo de acesso à memória nesta arquitetura é não uniforme pois depende do processador que realiza o acesso. Dependendo da distância entre o processador e o banco de memória que este pretende aceder, pode ser necessário realizar mais ou menos travessias pelos conetores para aceder ao banco de memória que contém os dados.

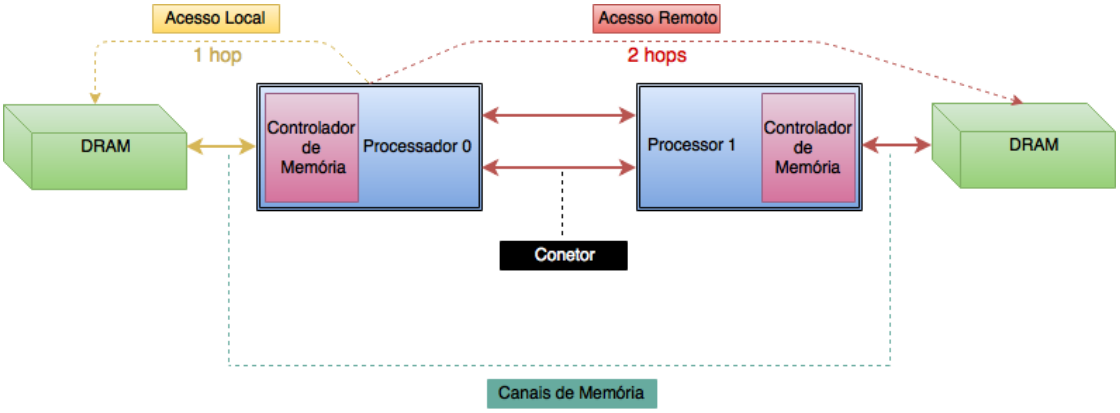


Figura 3.: Exemplo de uma arquitetura NUMA com dois processadores (2-socket).

Os processadores são ligados entre si através de conetores bi-direcionais (geralmente QPI[3] ou HyperTransport[4]) para permitir o acesso de cada processador aos nós NUMA dos restantes processadores. Arquiteturas NUMA mais recentes (p.e a família Intel Xeon Scalable[5]) usam um novo conector designado por UPI (*UltraPath Interconnect*) - o sucessor do QPI que existia nos processadores Intel desde 2007. Estes conetores foram desenhados para terem baixa latência e elevada largura de banda por forma a minimizar o tempo necessário para efetuar acessos remotos e permitir uma maior escalabilidade.

A figura 3 mostra uma arquitetura NUMA com dois processadores (geralmente designada de *dual-processor*). A seta amarela em tracejado que liga o processador 0 à memória que lhe está ligada diretamente pelo canal de memória representa um *acesso local*. Este é o termo utilizado sempre que um processador realiza um acesso ao seu próprio banco local de memória. No entanto, dependendo da política de alocação dos dados realizada pela aplicação ou do sistema operativo, o processador pode ter que aceder a dados de memória que estão alocados na DRAM de outro processador. Este tipo de acesso é designado por *acesso remoto* e está representado na figura 3 com a seta vermelha a tracejado.

3 QPI - QuickPath Interconnect é um conetor desenvolvido pela Intel introduzido nos processadores da arquitetura Nehalem em 2007.

4 HyperTransport é o nome do conetor ponto-a-ponto de baixa latência desenvolvido pela AMD para interligar os processadores (inicialmente na familia Opteron) e chipsets NVidia.

5 Comercializadas apartir de Julho de 2017.

2.3.3 *Interação entre sistema operativo, aplicações e a arquitetura NUMA*

O uso da arquitetura NUMA num sistema de computação exige uma versão do *kernel NUMA aware* por forma a que o sistema operativo consiga compreender a organização de memória, e consiga realizar as alocações de memória de forma eficiente. No sistema operativo Linux, o *kernel* é responsável pela interação com o escalonador de fios de execução e alocação de páginas de memória das aplicações[18].

Quando uma aplicação é executada, é criado um processo com um dado PID (identificador desse processo). Considerando uma aplicação paralela implementada na linguagem C, o processo irá criar tantos descendentes[6] como o número de fios de execução criados. Em algumas linguagens modernas como o Java, não é possível fazer esta identificação trivialmente. Quando uma aplicação Java é executada, esta corre sobre uma máquina virtual designada por *Java Virtual Machine (JVM)* que confere portabilidade e suporte multiplataforma mas que introduz uma nova camada sobre a execução do programa. Isto faz com que além dos fios de execução da aplicação, sejam também criados fios de execução adicionais inerentes à própria JVM. Em linguagens como o Java, não é possível ao programador controlar nativamente e mapear os fios de execução em núcleos de processamento específicos.

2.3.4 *Memória virtual e alocação de páginas em nós NUMA*

A memória virtual é um mecanismo que permite ao sistema operativo utilizar outros dispositivos de armazenamento como memória secundária expandindo assim a memória disponível. Desta forma, dados podem estar localizados em dois sítios distintos: em memória principal (física) ou num outro dispositivo. Ao nível da memória virtual existe o movimento de dados entre a memória principal e o armazenamento secundário (técnica é conhecida como *swapping*). Os sistemas operativos modernos introduziram uma forma mais eficiente e leve de lidar com a memória virtual designada por *paging*. Em vez de se mover processos inteiros entre as duas memórias apenas pequenos segmentos de memória são movidos (geralmente blocos de 4 KB em Linux).

6 Em sistemas operativos, também designados como processos *filho* (children) de um determinado processo pai.

Numa arquitetura NUMA é possível saber informação sobre como é que as alocações de memória estão a ser realizadas entre os nós NUMA. No caso do Linux, apartir da versão 2.6.14 do *kernel*[32] cada processo que corre numa arquitetura NUMA possui um ficheiro associado (**numa_maps**) localizado em */proc/<PID>/numa_maps*. Este ficheiro possui informação ao alcance do programador acerca das zonas de memória ocupadas pelo processo, e quais os nós NUMA escolhidos para mapear as páginas de memória relativas a esse processo. Um excerto das alocações de um processo é reproduzido abaixo:

```
1 ...
2 2b11e6153000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so mapped=13 N1=13
3 2b11e6160000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so
4 2b11e6360000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so anon=2 dirty=2 N1=2
5 2b11e6362000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so mapped=32 N1=32
6 2b11e638c000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so
7 2b11e658c000 default file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so anon=2 dirty=2 N1=2
8 ...
```

Cada nó NUMA é identificado pelo sistema operativo através de um identificador que é utilizado igualmente para identificar o processador num determinado socket. Assim, numa arquitetura NUMA equipada com 4 processadores, o primeiro processador e banco de memória são identificados com o valor 0, os segundos são identificados com o valor 1 e assim sucessivamente. O excerto apresentado corresponde a um excerto de um ficheiro *numa_maps* de um PID que corresponde a uma aplicação Java em execução. Cada linha corresponde a uma região de memória da aplicação. O primeiro campo corresponde ao endereço de memória onde começa a região de memória, e o segundo contém o nome da política de alocação de memória atualmente em uso. O ficheiro contém uma lista dos endereços de memória onde o endereço começa, mostra a política de alocação de memória utilizada, o número de páginas alocadas e uma referência que indica em qual dos nós NUMA essas páginas foram alocadas. A atual política de alocação de memória é aquela que está definida por omissão o que significa que o escalonador do kernel não foi instruído pela aplicação a alterar a política de alocação que determina a forma como as páginas de memória são distribuídas entre nós NUMA. O terceiro campo corresponde ao código presente nessa região de memória. O resto dos campos indicam o número de páginas alocadas (*mapped*) num determinado nó NUMA utilizando a sintaxe N<Nó>=<nrpáginas>. Na linha 5, a região de memória começa no endereço **2b11e63620** que possui 32 páginas de memória todas elas alocadas no nó 1 (N1). A aplicação corre numa arquitetura com dois processadores e uma vez que a

numeração atribuída aos nós é inteira e iniciada em 0, isto significa que as alocações no nó 1 referem-se a alocações na memória do segundo processador. Existe outro tipo informação ao longo das linhas destes endereços. Por exemplo, o número de páginas ativas, e o número de páginas que foram marcadas como sujas (*dirty*) e que precisam de ser movidas para disco antes de serem limpas.

No exemplo, as alocações estão a serem feitas no segundo nó NUMA. Se os fios de execução estiverem a correr no primeiro processador, essas alocações tornam necessário a realização de acessos remotos que implicam uma maior latência no acesso o que prejudica o desempenho da aplicação. Como será mostrado na secção 3.3, algumas ferramentas (por exemplo *numactl*) e bibliotecas de sistema podem ser usadas para instruir o *kernel* a alterar a política das alocações. Isto permite melhorar a afinidade dos fios de execução em relação aos dados por forma a aumentar o desempenho.

No presente exemplo, se o *numactl* for utilizado para forçar as alocações de memória no nó 0, é possível verificar uma diferença nos resultados no ficheiro numa_maps:

```
1 ...
2 2b3cbca65000 bind:0 anon=28 dirty=28 N0=28
3 2b3cbcb63000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so mapped=13 N1=13
4 2b3cbcb70000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so
5 2b3cbcd70000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so anon=2 dirty=2 N0=2
6 2b3cbcd72000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so mapped=32 N1=32
7 2b3cbcd9c000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so
8 2b3cbcf9c000 bind:0 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so anon=2 dirty=2 N0=2
9 ...
```

Em vez de utilizar alocações por omissão, agora o *kernel* modificou a política de alocação de memória por forma a realizar as alocações de páginas no nó 0 (N0) como definido com *numactl*. É possível verificar estas alterações nos ficheiros numa_maps dos processos correspondentes à execução da aplicação Java. O *numactl* pode ser também utilizado para intercalar as páginas de memória uniformemente entre nós NUMA com o uso da opção *interleave=<lista_de_nós>*. Nesse caso, o ficheiro numa_maps terá a tag *interleave* seguida dos identificadores dos nós NUMA pelos quais as páginas de memória são distribuídas:

```
1 ...
2 00400000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/bin/java mapped=1 N0=1
3 00600000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/bin/java anon=1 dirty=1 N0=1
4 01eaf000 interleave:0-1 heap anon=4 dirty=4 N0=2 N1=2
5 50b000000 interleave:0-1 anon=418816 dirty=418816 active=417280 N0=209408 N1=209408
6 5ab080000 interleave:0-1
7 6d9000000 interleave:0-1 anon=171904 dirty=171904 N0=85952 N1=85952
8 ...
9 2ae5d287e000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/server/libjvm.so N0=2127
10 2ae5d34e8000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/server/libjvm.so
```

```
11 2ae5d36e7000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/server/libjvm.so  N0=107 N1=108
12 2ae5d37be000 interleave:0-1 anon=48 dirty=48 N0=24 N1=24
13 ...
14 2ae5d3803000 interleave:0-1 anon=28 dirty=28 N0=14 N1=14
15 2ae5d3901000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so mapped=13 N0=13
16 2ae5d390e000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so
17 2ae5d3b0e000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libverify.so anon=2 dirty=2 N0=1 N1=1
18 2ae5d3b10000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so mapped=32 N0=32
19 2ae5d3b3a000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so
20 2ae5d3d3a000 interleave:0-1 file=/share/apps/java/jdk1.8.0_20/jre/lib/amd64/libjava.so anon=2 dirty=2 N0=1 N1=1
```

Os excertos mostrados até agora revelam que o escalonador do sistema operativo honra as alterações de alocamento das páginas com o uso da ferramenta numactl utilizada numa aplicação em Java. Note-se ainda que pelas linhas 5, 7, e 11, 12, 17 e 20 o número de páginas alocadas no nó 0 (N0) e nó 1 (N1) são praticamente as mesmas verificando-se uma distribuição equitativa das páginas pelos dois nós.

Além de ferramentas externas, ao nível da programação também existem chamadas ao sistema que podem ser utilizadas para controlar as alocações de memória. Esse estudo será feito mais a frente no capítulo 3.

2.3.5 *Acessos locais e acessos remotos: Impacto no desempenho*

Para fazer uma análise de desempenho em arquiteturas NUMA, torna-se conveniente medir o número de acessos locais e remotos em relação ao número de acessos de memória totais. Os acessos de memória por um processador realizados à sua memória local são acessos de menor latência e por isso, mais rápidos que os acessos remotos. Os acessos remotos envolvem o acesso a dados à memória principal de processadores mais distantes na arquitetura o que implica o transporte de dados através de conectores que interligam os vários processadores. A contagem do número de acessos locais e remotos pode ser útil para perceber se é possível aumentar o desempenho minimizando o número de acessos remotos e maximizando acessos locais. O número de acessos locais e remotos de uma aplicação em Java podem ser contabilizados com recurso a ferramentas de *profilling* baseadas em contadores de hardware (p.e *PAPI* para C, *PAPIJ*[7] para aplicações em Java, ou o *Processor Counter Monitor (PCM)[16]* para C ou Java).

7 O PAPIJ é uma ferramenta que permite a utilização da ferramenta PAPI em Java. Consiste numa chamada em Java ao PAPI com utilização de JNI. A ferramenta foi criada pelo grupo de investigação de Arquitetura de Computadores da Universidade do Minho.

2.3.6 *A métrica Hop*

Em certos estudos realizados sobre NUMA (p.e [45]) também é usual a adopção de métricas baseadas no número de conetores e barramentos implicados num acesso a dados na DRAM: **Hop**. A métrica hop representa o número de conectores e barramentos que um processador precisa de atravessar para satisfazer um acesso (local ou remoto) de memória principal [18].

Considere-se novamente a figura 3 apresentada anteriormente. Num acesso local, o Processador 0 na figura interroga o endereço de memória ao seu próprio controlador de memória (rosa) e apenas um barramento de memória é atravessado para aceder aos dados da memória principal. Este acesso representa concretamente 1 *hop*. Por outro lado, se este processador realizar um acesso remoto, há um maior esforço envolvido. Um pedido que não pode ser satisfeito localmente, tem que percorrer o conector a vermelho para chegar ao processador vizinho, fazer o pedido ao respetivo controlador de memória e atravessar posteriormente o canal de memória para aceder a essa memória. No processo foi necessário atravessar dois conetores pelo que este acesso remoto envolveu 2 hops (o dobro do necessário no acesso local). Nesta análise, verifica-se que existe um novo desafio de otimização das aplicações na arquitetura NUMA que passa por reduzir o número de hops por acesso à DRAM tirando partido do fato de cada processador ter memória local ligada diretamente a si. A diminuição do número de *hops* por acesso, ajuda a maximizar acessos locais e minimizar acessos remotos.

Acessos remotos prejudicam o desempenho por causa do tráfego entre conectores e barramentos de memória implicados. Além disso, quantos mais nós NUMA existirem na arquitetura, maior é o número de barramentos utilizados para interligar os processadores. Como consequência, cada acesso remoto poderá implicar um maior número de *hops* necessário para realizar esse acesso. Basta para tal considerar uma arquitetura NUMA que em vez de utilizar 2 processadores (como na figura 3), utiliza 4 ou até 8 processadores interligados como se verifica em arquiteturas mais recentes.

2.3.7 *Configurações NUMA avançadas: 4-socket e 8-socket*

A arquitetura da figura 4 possui uma organização de memória igual aquela que pode ser encontrada nas configurações Intel 4S da familia *Xeon Scalable* da micro-arquitetura Skylake[10], ou na familia *Intel Xeon E7*. Nesta figura é possível distinguir dois tipos distintos de configurações: uma configuração em anel (*ring*) e a de inclusão de conetores cruzados (*crossbar configuration*). Numa configuração em anel, os processadores são interligados pelos conectores assinalados a preto na figura pelo que o processador 0 e o processador 3 não estão ligados entre si tal como acontece entre o processador 1 e 2. A configuração *crossbar* por sua vez é composta pelos conectores a preto, e adicionalmente pelos conetores a vermelho por forma a aumentar o grau de interligação entre os processadores o que permite que cada processador seja adjacente dos restantes. Esta característica pode reduzir 1 hop por cada acesso remoto que envolva transporte de dados entre processadores com posição cruzada na arquitetura 4-socket.

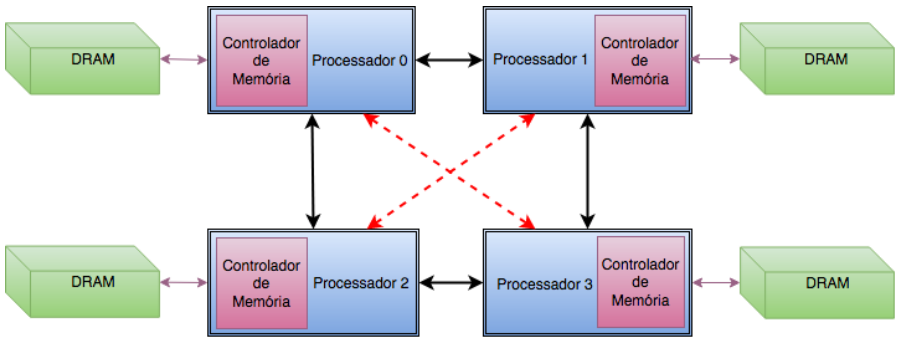


Figura 4.: Exemplo de arquitetura NUMA com 4 processadores (4-socket)

Comparando a configuração 2-socket e 4-socket NUMA (na figura 3), as diferenças mais óbvias estão no facto de se ter duplicado o número processadores e bancos de memória principal, assim como o número de conetores entre processadores aumentou consideravelmente. A duplicação de recursos (processador + banco de memória) pressupõe um ganho adicional teórico de 2 vezes em largura de banda. Contudo, pela mesma figura é possível verificar que a duplicação dos recursos implica a necessidade de mais comunicação entre processadores sendo que na configuração 2-socket existe apenas um conetor bidirecional (geralmente com 2 ligações), e na configuração 4-socket existem até 6 conetores (na configuração *crossbar*). Ao contrário do que se verifica numa ar-

quitetura com 2-sockets em que um banco de memória pode ser acedido pelos núcleos de 2 processadores (o local e o processador vizinho), numa arquitetura 4-socket, cada controlador de memória pode ser acedido através dos núcleos de processamento de até 4 processadores distintos fazendo com que os acessos remotos possam ter uma maior latência.

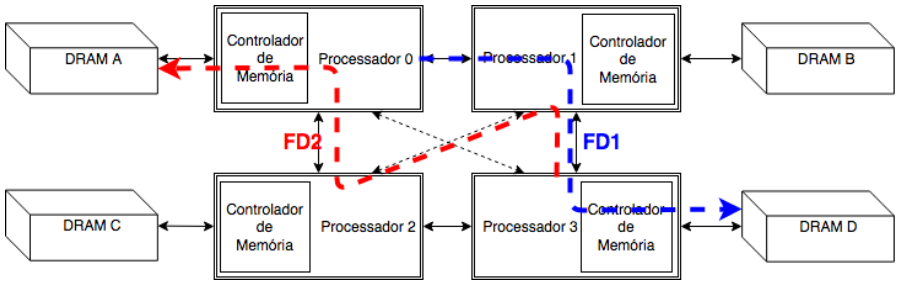


Figura 5.: Exemplo de dois possíveis fluxos de dados em dois exemplos de acessos remotos numa arquitetura 4-socket com configuração *crossbar*

A figura 5 ilustra exemplos em que determinados fluxos de dados que realizam acessos remotos. Estes acessos podem implicar penalizações de desempenho cada vez maiores à medida que se aumentam o número de nós NUMA.

O fluxo de dados a azul (FD1) representa um acesso remoto do processador 0 à *DRAM D* do processador 3. Cada acesso resulta numa travessia de 3 *hops* - um *hop* adicional comparativamente com os acessos remotos numa arquitetura 2-socket. No caso do segundo fluxo de dados (FD2) a travessia é de 4 *hops* - o dobro das travessias por cada acesso remoto.

Por um lado, numa configuração 2-socket cada acesso remoto sofre a penalização de 2 *hops* por acesso remoto. Por outro lado, numa arquitetura 4-socket existe o dobro de nós NUMA, o dobro da largura de banda teórica, mas os acessos remotos podem sofrer uma dupla penalização. A somar a essa penalização que resulta em latências maiores por cada acesso, à medida que se aumentam o número de nós NUMA é de esperar uma maior sobrecarga de comunicação entre processadores. Esta preocupação aumenta se considerarmos uma arquitetura 8-socket:

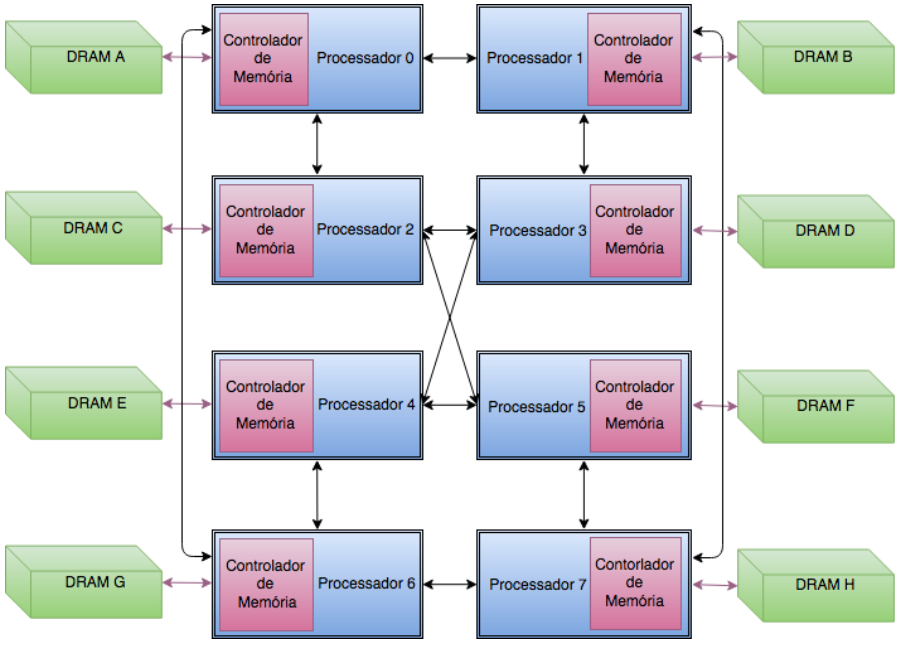


Figura 6.: Exemplo de uma arquitetura NUMA com 8 processadores (8-socket)

O aumento sucessivo do número de processadores fez com que se aumentasse o número de conetores por forma a maximizar a conexão direta dos processadores às memórias remotas. Principalmente, a ligação entre processadores mais distantes que sem conetores diretos sofrem penalizações de acessos remotos maiores.

2.3.8 *Conetores de interligação de processadores (interconnects)*

Os conetores que interligam os processadores nas arquiteturas NUMA são, como constatado anteriormente, uma limitação de desempenho nas arquiteturas quando o número de acessos remotos aumenta significativamente. Considerando as arquiteturas NUMA dos últimos anos, é comum verificar a existência de conetores *Quick Path Interconnect* com um máximo de largura de banda anunciada de 12.8 GB/s unidirecional o que resulta num total de 25.6 GB/s[33]. No entanto, os acessos de memória locais uma largura de banda disponível superior à dos conetores. Os acessos locais possuem uma largura de banda anunciada de 59.7 GB/s (num processador *Intel Xeon E5-2650 v2*[6]) e 76.8 GB/s (no modelo E5-2687W v4 [29]). Os canais de memória local possuem então uma largura de banda superior. Nos acessos remotos, mesmo que o nó remoto tenha

igualmente uma elevada largura de banda, a taxa de transferência de dados a transportar está limitada pela largura de banda do conector. Em alguns processadores como o *Intel Xeon E5-2687W v4* foi melhorada a tecnologia do conetor chegando a uma largura de banda de 9.6 GT/s com QPI que equivale a uma largura de banda unidirecional de 19.2 GB/s teórica[29] e um máximo total de 38.4 GB/s (bidirecional).

Recentemente, alguns esforços têm sido feitos pelos fabricantes em novas arquiteturas NUMA em introduzir novas tecnologias de conetores. O objectivo é dotar os processadores destas arquiteturas com conetores de uma largura de banda superiores às que existiam antes nos conectores QPI, e igualmente para versões mais antigas do HyperTransport nas arquiteturas AMD. Larguras de banda superiores fazem com que se consiga transportar mais dados por unidade de tempo. A última versão do *HyperTransport* (versão 3.1), suporta um máximo de 51.2GB/s de largura de banda, e a Intel desenvolveu recentemente o *Intel Ultra Path Interconnect (UPI)* - o sucessor do QPI com uma largura de banda máxima anunciada de 10.4 GT/s[10], o que corresponde a uma largura de banda máxima por UPI de 41.6 GB/s (contra 38.4 GB/s através de QPI). Esta evolução permitiu reduzir a assimetria de larguras de banda entre memória local e largura de banda do QPI por forma a minimizar o custo de acessos remotos não sendo contudo o suficiente para eliminar este *bottleneck*.

Nesta dissertação será constatado, no entanto, que apesar das diferenças de largura de banda entre canais de acesso direto à memória e conetores de interligação de processadores, a realização de um elevado número de acessos remotos não é o principal fator que limita o desempenho. O aspeto verdadeiramente importante no aumento de desempenho das aplicações nestas arquiteturas está relacionado com um aproveitamento eficiente da largura de banda utilizada à DRAM.

3

DESEMPENHO DE APLICAÇÕES EM ARQUITETURAS NUMA

3.1 INTRODUÇÃO

Atualmente, existe um aumento do número de algoritmos cujo desempenho está limitado pelos componentes de memória devido às necessidades intensivas de acesso a grandes quantidades de dados (algoritmos *memory-bound*). A exploração de paralelismo ao nível das instruções chegou a um limite que levou os fabricantes a focar atenção no paralelismo ao nível dos fios de execução[11]. Ao longo dos anos as linguagens imperativas como Fortan e C/C++ passaram a incluir suporte para paralelismo ao nível dos fios de execução. Este suporte é baseado em bibliotecas e/ou construtores baseados em diretivas que conseguem tirar partido de ambientes de memória partilhada. Paralelizando uma aplicação é possível dividir a carga computacional de uma aplicação em fios de execução. Cada um desses fios pode ser processado em paralelo em diferentes núcleos no processador onde a aplicação começou a correr (local) ou nos núcleos de processadores vizinhos.

Nas aplicações paralelas, por vezes, os fios de execução e os processos acedem a memória remota originando eventuais perdas de desempenho. Por outro lado, os acessos remotos podem ser usados para aumentar a largura de banda utilizada porque além do banco de memória local, está a ser utilizado simultaneamente um banco de memória remota (com eventual penalização de atravessar os conetores de interligação dos processadores). Estas aplicações devem ter em consideração a afinidade dos fios de execução e a afinidade dos dados. Isto porque existem múltiplas configurações possíveis de afinidade dos fios de execução e dados que resultam em padrões diferentes de acessos à memória, originando diferentes níveis de desempenho com diferentes penalizações as-

sociadas nas arquiteturas NUMA. O impacto da afinidade dos fios de execução e dos dados será estudada e aplicada em mais detalhe na componente experimental deste estudo no capítulo 4.

A linguagem Java por ser uma linguagem portável e segura é uma linguagem muito utilizada. Por essa razão têm crescido as preocupações relacionadas com o aumento de desempenho. O Java oferece suporte nativo para aplicações paralelas com o uso da classe *Thread* nativa do Java em conjunto com a interface *Runnable* e métodos de manipulação de fios de execução.

Neste capítulo serão exploradas ferramentas, mecanismos e técnicas de programação que permitem explorar a afinidade de fios de execução e de dados. Será especificamente explorado o suporte para aplicações em Java fazendo uma comparação com os resultados obtidos em aplicações em C. Este estudo constitui um dos principais focos da dissertação visando aumentar o desempenho destas aplicações em arquiteturas NUMA.

3.2 AFINIDADE DE FIOS DE EXECUÇÃO E DADOS

Por forma a aumentar o desempenho de um determinado algoritmo, a configuração da distribuição de fios de execução entre os núcleos e as alocações de memória devem ser tidos em consideração.

É possível mapear os fios de execução em núcleos de processamento de diversas formas. É possível distribuir os fios de execução de forma intercalada pelos núcleos, distribuir mais que um fio de execução por núcleo, limitar o mapeamento só em núcleos físicos (não utilizando os núcleos lógicos), mapear pelos núcleos do primeiro processador, entre outros. A forma como estes mapeamentos são feitos dá origem a padrões de acesso à memória principal distintos. Isto conduz-nos ao conceito de afinidade de fios de execução[11]. O termo *binding* (mapeamento) é também usado para descrever o processo de distribuir fios de execução por núcleos de processamento específicos[11].

Nas arquiteturas NUMA, os fabricantes tem equipado os processadores com hierarquias de memória com múltiplos níveis de *cache* privada (tipicamente L1 e L2) em cada núcleo, e níveis de *cache* partilhada (L3) assim como múltiplos bancos de memória (um por nó). Os diferentes níveis de *cache* assumem um papel importante na hierarquia de

memória por forma a evitar latências de acesso a memória DRAM. A distância entre cada fio de execução e os dados também tem uma implicação direta no desempenho que deriva do facto dos acessos serem realizados de forma não-uniforme conduzindo ao conceito de afinidade de dados. Estes caracterizam a distância relativa dos dados de memória em relação ao processador que acede a esses dados. Deste modo, existe uma relação direta entre os conceitos de afinidade de dados e de fios de execução. Se os fios de execução de uma dada aplicação foram mapeados nos núcleos de um certo processador mas estes precisarem de aceder a dados de memória localizadas numa DRAM remota, é de esperar um desempenho inferior em comparação com um padrão de acesso em que os fios de execução acedem aos mesmos dados mas armazenados na DRAM local (mais próxima). Dependendo da afinidade estabelecida, os fios de execução podem estar mais próximos ou mais afastados dos dados que estes precisam de aceder. A manipulação de diferentes configurações de afinidade permite afinar o padrão de acesso de fios de execução a dados em memória principal por forma a escolher aquela que oferece o melhor desempenho possível numa arquitetura NUMA.

3.3 ABORDAGENS DE CONTROLO DE AFINIDADE

Regra geral, nas aplicações em Java, o *kernel* tenta respeitar as alterações ao nível das políticas de alocação de memória (distribuição de páginas de memória) realizadas de forma explícita pelo programador com recurso a ferramentas externas.

Este estudo explora um conjunto de opções de compilação, ferramentas externas como *numactl* e API's que aplicam diferentes tipos de afinidade de fios de execução e dados de aplicações em Java e C. Cada sistema operativo tem o seu próprio escalonador que permite ao *kernel* atribuir fios de execução aos núcleos de acordo com as suas heurísticas de escalonamento. Contudo, estas heurísticas podem não garantir o melhor desempenho por não considerarem a arquitetura NUMA do sistema pelo que diferentes configurações de afinidade de fios de execução e dados podem ter um impacto considerável no desempenho das aplicações.

Quando a estratégia de afinidade é alterada, a política de alocação do escalonador do sistema operativo é redefinida. Isto pode ocorrer ao nível das alocações de páginas de memória, ou ao nível dos fios de execução. O *kernel* tenta minimizar a migração

de fios de execução para núcleos de processamento diferentes mas isto nem sempre tal é garantido[45]. A ocorrência de migração de fios de execução (*thread-migration*) em certas tarefas é frequente e realizada automaticamente pelo escalonador do sistema operativo[45]. Por exemplo, se o escalonador do sistema operativo mapear os fios de execução num determinado núcleo de processamento e alguns destes estiverem em estado de espera, o escalonador pode migrá-los para outros núcleos.

Uma boa estratégia de afinidade é importante para permitir uma maior escalabilidade dos algoritmos através da distribuição de fios de execução e alocações de memória, que são aspetos críticos em NUMA. Nestas arquiteturas é importante garantir que depois dos fios de execução serem alocados, estes não sofrem de migração para outros núcleos de processamento. As migrações entre nós aumentam o tráfego nos conetores e degradam o desempenho. Por essa razão, as ferramentas de *binding* de fios de execução podem ter uma especial importância ao fixar o mapeamento realizado diminuindo a ocorrência de migrações de fios de execução.

O alcance deste estudo será restringido aos mecanismos de afinidade que possam ser utilizadas em aplicações Java e C. Para a linguagem C/C++, a afinidade de fios de execução pode ser conseguida através de:

- Ambientes de programação que implementam paralelismo (i.e *OpenMP* ou *POSIX* threads)
- Variáveis de afinidade como a *GOMP_CPU_AFFINITY* e *KMP_AFFINITY* para o compilador GCC da GNU e ICC da Intel, respetivamente.
- Bibliotecas de afinidade (*Java Thread Afinity Library* e *CoralThreads*);
- Ferramentas do sistema operativo. Por exemplo, *taskset* em Linux (ou *pbind* no Solaris), *numactl* e *migratepages*;
- Chamadas a sistema (por exemplo *sched_setaffinity* ou *sched_getaffinity*);

Para Java, a afinidade de fios de execução não é suportada nativamente. No entanto existem API's externas tais como *Java Thread Library*[1] e *CoralThreads*[5] que permitem aumentar o grau de suporte.

3.3.1 *OpenMP*

O *OpenMP* é uma API baseada em diretivas desenvolvida para arquiteturas de memória partilhada que permite explorar paralelismo em aplicações escritas em C, C++ e Fortran. A API permite utilizar diretivas que dividem a carga de processamento em subconjuntos mais pequenos que possam ser processados em paralelo por uma equipa de fios de execução. O número de fios de execução e a afinidade dos mesmos podem ser definidos utilizando as cláusulas *num_thread* e *proc_bind* respectivamente. A cláusula *proc_bind* têm três estratégias de afinidade distintas:

- **master**: Coloca cada fio de execução no mesmo *place*[1] do fio de execução master.
- **spread**: Espalha os fios de execução entre diferentes *places*.
- **close**: Os fios de execução são mapeados próximos do fio de execução mestre.

Um exemplo de utilização da diretiva *proc_bind* seria o seguinte:

```
1 #pragma omp parallel proc_bind(spread) num_threads(8)
2 {
3     doParallelWork();
4 }
```

Neste caso, 8 fios de execução são criados e espalhados por diferentes núcleos de processamento. As estratégias de afinidade podem igualmente ser definidas antes de executar a aplicação exportando a variável de ambiente OMP_PROC_BIND com a afinidade desejada (por exemplo, OMP_PROC_BIND=spread). O mesmo se aplica à variável OMP_PLACES para redefinir o alcance de cada *place*. Enquanto que a variável OMP_PROC_BIND estabelece a forma como os fios de execução são escalonadas nos diferentes *places*, a variável OMP_PLACES define os *places* no hardware aos quais os fios de execução são atribuídos. Os valores possíveis para OMP_PLACES são *cores*, *sockets* ou *threads*.

Considere-se uma arquitetura NUMA com dois nós com o total de 16 núcleos de processamento. Se a variável OMP_PLACES for redefinida com OMP_PLACES=sockets,

1 Um *place* é criado geralmente por núcleo a menos que a varivável OMP_PLACES seja definida com um valor específico que altera o alcance de cada *place* (sockets ou fios de execução)[3]

os fios de execução são mapeados de forma intercalada entre os sockets disponíveis[41]. A figura 7 representa este tipo de *binding*. O bancos e os canais de memória foram removidos para simplificar a figura.

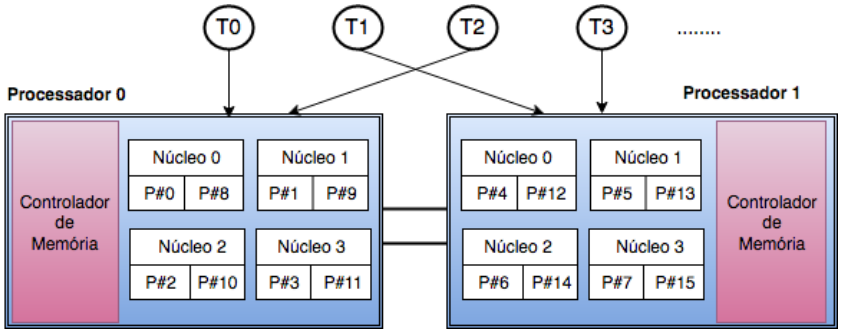


Figura 7.: Mapeamento de fios de execução se OMP_PLACES=sockets for definido.

Caso se pretenda as afinidades *close* ou *spread*, então a variável OMP_PROC_BIND pode ser utilizada para expressar essas afinidades sendo o resultado desses mapeamentos ilustrados na figura 8 e 9:

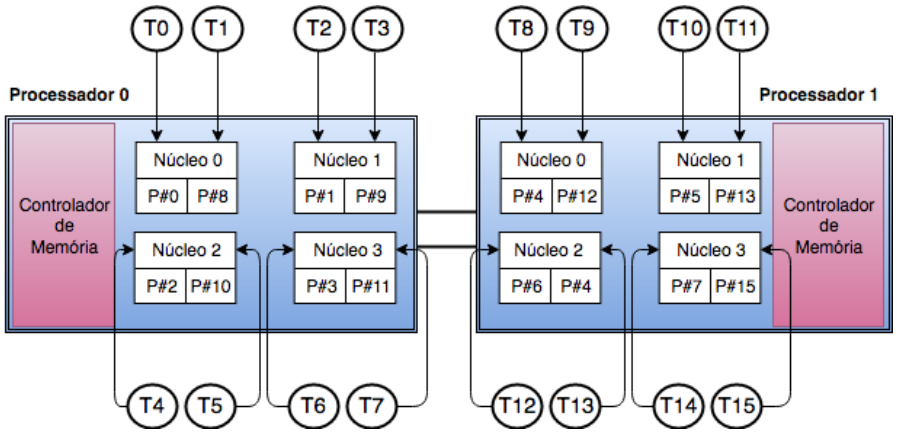


Figura 8.: Estratégia de mapeamento de fios de execução caso seja definido OMP_PLACES=cores e OMP_PROC_BIND=close.

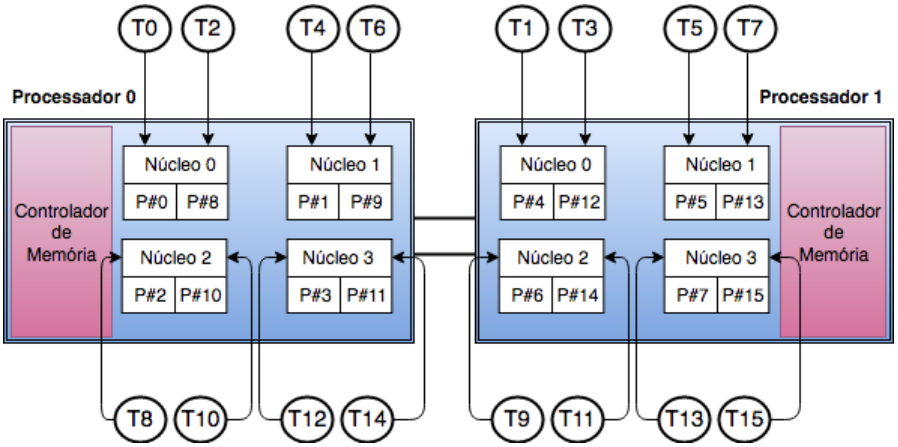


Figura 9.: Estratégia de *binding* com OMP_PLACES=cores e OMP_PROC_BIND=spread.

É recomendado ativar OMP_DISPLAY_ENV com o valor *true* por forma a informações sobre as estratégias de afinidade atuais sejam imprimidas [41]. Infelizmente não é imprimida informação sobre o mapeamento que é feito dos fios de execução com OMP_DISPLAY_ENV mas é possível intrumentar a aplicação com recurso às funções *omp_get_thread_num()* e *sched_getcpu()* dentro da região paralela por forma a obter o ID do fio de execução e do núcleo onde cada fio de execução está a correr.

3.3.2 *Controlo da afinidade com variáveis e opções de afinidade dos compiladores*

A próxima abordagem de controlo de afinidade passa por utilizar variáveis de ambiente dos compiladores. O compilador *Intel C compiler (ICC)* e o *GNU C Compiler* providenciam as variáveis KMP_AFFINTIY e GOMP_CPU_AFFINITY, respectivamente. Como abordado anteriormente na secção 3.3.1, as variáveis OMP_PLACES e OMP_PROC_BIND permitem escolher entre algumas estratégias de afinidade mas as distribuíções dos fios de execução já são pré-definidas. A maior vantagem das variáveis e opções de afinidade disponíveis pelos compiladores da Intel e GNU está na possibilidade de poder fazer o mapeamento explicitamente.

Com a variável KMP_AFFINITY estão disponíveis as seguintes estratégias para fazer o mapeamento:

- **none**: Não será realizado o *binding*, o que não significa que API's como o *OpenMP* não o faça caso o sistema operativo e o hardware o suportem. Este é o *binding* definido por omissão.

- **disable**: Desativa por completo a afinidade de fios de execução mesmo que o sistema operativo a suporte.

- **balanced**: Este é um tipo de afinidade apenas disponível em processadores *Intel Xeon Phi*. Os fios de execução são distribuídos primeiramente pelos núcleos fisicos. Quando cada núcleo tiver pelo menos um fio de execução, uma nova ronda de fios de execução é distribuída pelos núcleos.

- **scatter**: Espalha os fios de execução pelos processadores numa lógica *round-robin*. Esta estratégia oferece geralmente um bom desempenho ao fazer uma distribuição alternada dos fios de execução entre os processadores[2].

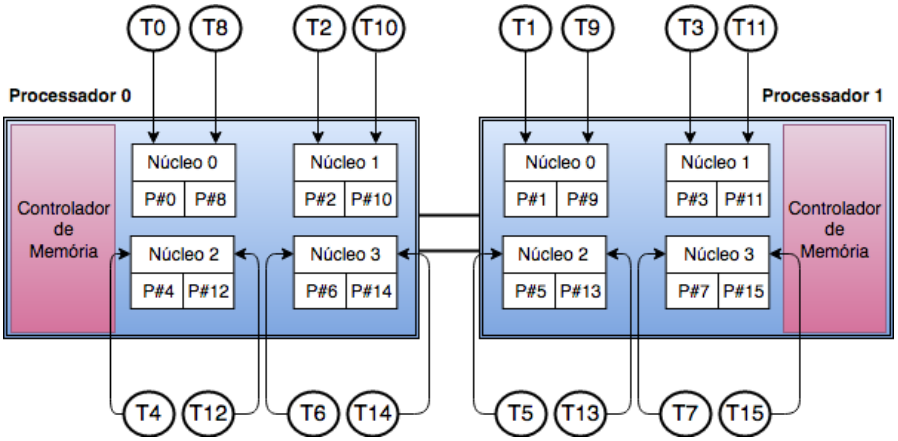


Figura 10.: Mapeamento de 16 fios de execução com uso de KMP_AFFINITY *scatter* numa arquitetura NUMA com 16 *núcleos*

- **compact**: Esta estratégia resulta numa maior afinidade de fios de execução por núcleo uma vez que a prioridade é preencher cada núcleo com fios de execução antes de mapear no núcleo seguinte do mesmo processador. Em arquiteturas *many-core* esta política pode levar a um desaproveitamento de utilização dos núcleos de processamento se existir um número insuficiente de fios de execução para preencher todas as unidades de processamento.

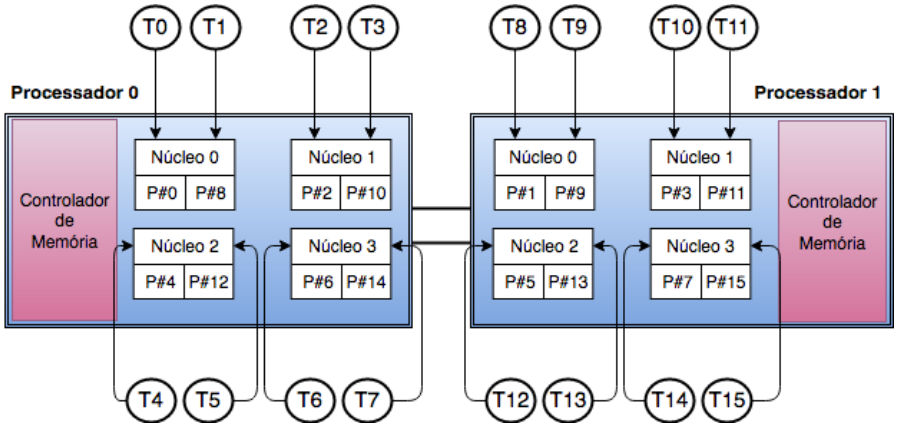


Figura 11.: Mapeamento de 16 fios de execução utilizando KMP_AFFINITY *compact* numa arquitetura NUMA com 16 *núcleos*

- **explicit**: Os fios de execução são mapeados nos núcleos pela ordem em que aparecem num vetor *proclist*, um fio de execução por núcleo. Considerando o ID do fio de execução como índices do vetor, os valores que aparecem no vetor proclist são os núcleos onde fios de execução serão mapeados. Este é o tipo de afinidade que oferece uma maior flexibilidade sobre o controlo de mapeamento dos fios de execução. Se for considerada a arquitetura 2-socket apresentada anteriormente e a variável KMP_AFFINITY é definida com execução de 8 fios de execução da seguinte forma:

```
1 KMP_AFFINITY=verbose,proclist=[0,4,1,5,2,6,3,7],explicit
```

  - O fio de execução 0 irá correr no núcleo 0;
  - O fio de execução 1 irá correr no núcleo 4;
  - O fio de execução 2 irá correr no núcleo 1 e assim por diante.

  Os mapeamentos manuais são geralmente respeitados pelo escalonador do *kernel* mas não é garantido que a aplicação não possa sofrer de migração de fios de execução durante a execução.

  O modo *explicit* pode ser também utilizado para criar *places*, sendo que um determinado fio de execução pode ser mapeado num dos núcleos do *place* correspondente. Por exemplo, se 8 fios de execução forem executados:

```
1 KMP_AFFINITY=verbose,proclist
    =[0,2,4,6,{8,10},{8,10},{8,10,12,14},{8,10,12,14}],explicit
```

A estratégia de *binding* anterior torna possível que:

- Os fios de execução 0, 1, 2 e 3 corram no núcleo 0, 2, 4, e 6 respectivamente;
- Os fios de execução 4 e 5 serão mapeadas para correr nos núcleos 8 ou 10;
- E os fios de execução 6 e 7 irão correr nos núcleos 8, 10, 12 ou 14.

Na variável do compilador GNU C Compiler (gcc) equivalente não existe estratégias de afinidade pré-definidas como a *compact* ou *scatter* da variável *KMP_AFFINITY*. Todos os mapeamentos tem que ser especificados numa lógica semelhante ao do vetor de núcleos como o *proclist* da variável KMP_AFFINITY. A variável de afinidade GOMP_CPU_AFFINITY deve ser utilizada com cautela quando utilizada em conjunto com outras variáveis de afinidade (e.g *OpenMP*) sendo que quando a variável do *OpenMP* OMP_PROC_BIND é configurada, a variável GOMP_CPU_AFFINITY é ignorada devido à ordem de precedência.

Existem *flags* para ativar opções de afinidade no caso do compilador ICC da Intel. Por exemplo, a *-par-affinity* pode ser usada para definir as estratégias de afinidade[2]. Esta *flag* precisa também das *flags -parallel* ou *-Qopenmp* caso contrário a configuração de afinidade não terá efeito. Se a variável KMP_AFFINITY for definida, a opção *-par-affinity* terá maior precedência. Os valores possíveis para a opção *-par-affinity* são: *scatter, compact, none, disable,* e *explicit* já abordados anteriormente.

3.3.3 *Bibliotecas de afinidade de fios de execução para Java*

A afinidade de fios de execução embora não seja suportada nativamente, pode ser controlada com o uso da biblioteca *Java Thread Affinity Library*. Esta biblioteca é uma biblioteca *OpenSource* criada pela *OpenHFT*[2] e que com recurso ao *Java Native Access*, permite aceder a funções nativas de controlo de fios de execução e fazer o *binding* a núcleos de processamento específicos de uma forma declarativa. Existe igualmente

2 Disponível em: https://github.com/OpenHFT/Java-Thread-Affinity

uma biblioteca comercial desenvolvida pela *CoralBlocks*[3] que permite mapear os fios de execução em núcleos específicos.

A utilização de bibliotecas implicam no entanto a modificação do código Java original para criação das afinidades e consequentemente um maior esforço de programação.

3.3.4 *Ferramentas do Sistema Operativo*

taskset em Java

O *taskset* é uma ferramenta de Linux que permite fazer alteração de mapeamentos que, segundo a literatura, são respeitados pelo escalonador do *kernel*[18].

O taskset utiliza máscaras no formato hexadecimal para identificar processadores. Por exemplo, 0x00000001 refere-se ao núcleo 0, 0x00000003 refere-se ao núcleo 0 e 1 e assim por diante até 0xFFFFFFFF que se refere à utilização de todos os núcleos[18].

A forma mais simples de mapear fios de execução nos núcleos é utilizando o *switch* **-c** que permite especificar o inteiro que identifica a unidade de processamento em vez da máscara do núcleo. Considerando uma aplicação Java *multithread* que recebe como argumento o número de fios de execução criados com a classe *Thread*, através desta invocação do taskset:

```
1    taskset -c 0,2,4,6,8,10,12,14 java -jar myapp.jar 8
```

A ferramenta irá mapear os fios de execução que foram criado para correr *myapp.jar* nos núcleos referidos na lista que vem a seguir ao *switch*.

Em certas linguagens de programação, o código é compilado para código nativo e linkado com as bibliotecas necessárias para produzir um executável. Ao contrário destas linguagens, em Java o código é compilado para *bytecode* que é interpretado por uma máquina virtual de Java (*Java Virtual Machine ou JVM*). O uso da máquina virtual é um bom aspeto da linguagem porque oferece portabilidade com suporte multiplataforma. No entanto, o seu uso remove algum controlo sobre a execução do programa no hardware. É o que acontece no que toca à realização do *binding* de fios de execução para

3 Disponível em: http://www.coralblocks.com/index.php/getting-started-with-coralthreads/

núcleos especificos de processamento. Os fios de execução são criados na JVM e o controlo sobre o mapeamento desses fios de execução é decidido pela JVM. O destino desses mapeamentos são realizados de acordo com heurísticas próprias, não acessíveis ao programador e decididas pelo sistema operativo. No entanto, é possível alterar esses mapeamentos com o uso de ferramentas externas. Por exemplo, a ferramenta **taskset**, permite controlar corretamente esses fios de execução, porque essas alterações de mapeamentos são realizadas pelo sistema operativo.

numactl e Java

As ferramentas apresentadas até agora ajudam o programador a controlar apenas a afinidade de fios de execução. No entanto, ter o controlo sobre as alocações dos dados em memória é igualmente importante nas arquiteturas NUMA porque dependendo da localidade dos dados em relação aos fios de execução é possível minimizar acessos remotos. A conjugação de afinidades entre fios de execução e os dados, ajudam a afinar o ambiente de execução até obter um desempenho mais próximo do desempenho ótimo.

Por forma a evitar acessos remotos, os dados de memória devem estar o mais próximo possível dos fios de execução que acedem a esses dados. Geralmente, nas arquiteturas NUMA as páginas de memória são alocadas no nó onde o processador iniciou a sua execução. Isto conduz a uma perda de desempenho devido à latência do conetor que é necessário percorrer para aceder aos dados. No entanto, foi concluído no capítulo 4 que a travessia do conetor de interligação dos processadores não é a principal razão que conduz à perda de desempenho. Os acessos remotos devem ser evitados (uma vez que a largura de banda da memória local é superior à largura do conetor que é necessário atravessar) mas, o aspecto que influencia mais o desempenho é o aproveitamento que se consegue fazer da largura de banda utilizada.

Por forma a prevenir acessos remotos, são sugeridas duas configurações de afinidade (fios de execução e dados) possíveis nas figuras 12 e 13:

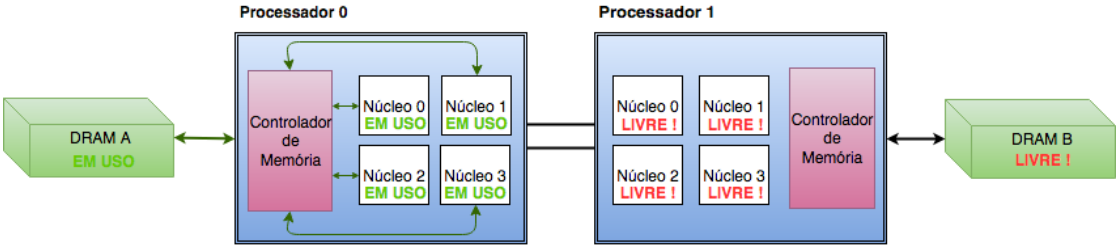


Figura 12.: Maximização de acessos locais realizando as alocações de fios de execução e respetivos dados no primeiro nó.

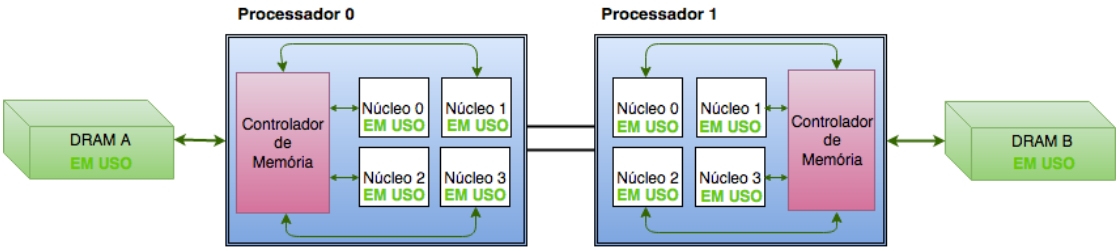


Figura 13.: Maximização de acessos locais mapeando os fios de execução e alocando os dados nos bancos de memória respetivos (sem tráfego no conector).

Na primeira figura, os fios de execução e os dados são alocados no primeiro nó NUMA pelo que apenas o banco de memória local (DRAM A) está a ser utilizado. Considere-se que a DRAM A possui capacidade de armazenamento para os dados da aplicação sem necessidade de usar a DRAM B. Este mapeamento evita o tráfego no conector. É expectável que um aumento do número de fios de execução aumente a quantidade de dados que é possível processar em paralelo. Contudo, à medida que se aumenta o número de fios de execução, aumenta-se a contenção no controlador de memória do primeiro nó para aceder a dados na DRAM A. Quando o número de fios de execução aumenta consideravelmente, pode ser então preferível distribuí-los pelo processador 0 até a um número ótimo (geralmente um fio de execução por núcleo) e mapear os restantes no processador 1. Contudo, os fios de execução no processador 1 podem precisar de aceder a páginas de memória da DRAM A, acessos esses que devem ser evitados.

Assim, deve ser considerada uma estratégia mais balanceada de forma a que os fios de execução sejam criados e executados nos processadores dos nós onde estão localizados

os dados que estes irão aceder. Esta é a estratégia utilizada na figura 13. Nesta figura, os fios de execução são criados e executados em cada um dos processadores, divididos de forma equitativa. Esta estratégia permite evitar o tráfego no conector, e ainda duplicar a largura de banda usada com a utilização dos dois bancos de memória. Assumindo que todos os fios de execução são mapeadas nos respetivos núcleos e estes não sofrem migração, esta deverá será a melhor estratégia de *binding* para arquiteturas NUMA.

A ferramenta *numactl* disponível para Linux, compatível com aplicações Java e C utilizada para explorar múltiplos fios de execução e dados em simultâneo. É possível restringir a execução das aplicações a núcleos específicos com o uso das opções *–cpunodebind=<lista_de_proc>* e *–physcpubind=<lista_de_núcleos>*. A lista de processadores corresponde a uma lista com os inteiros que identificam cada processador (isto é, cada *socket* com um processador multinúcleo). Os processadores à semelhança dos núcleos, são identificados através de inteiros atribuídos sequencialmente. Por exemplo, se for definida a opção *–cpunodebind=0,3* significa que a execução do programa será restringida aos processadores no socket 0 e no socket 3. A lista de núcleos é a lista que identifica as unidades de processamento que serão utilizadas durante a execução, separados por vírgulas. Por exemplo, se o *numactl* for executado com a opção *–physcpubind=0,8,1,9,2,10,3,11* numa configuração NUMA 2-socket como o da figura 8, a execução será restringida à utilização dos núcleos do processador 0 simulando uma estratégia de binding similar à estratégia *close* do KMP_AFFINITY.

As alocações de memória podem igualmente serem restringidas com o numactl em nós NUMA específicos com a utilização da opção *–membind=<lista_de_nós>*.

As páginas (ou segmentos) de memória podem ser intercalados entre os nós com o uso da opção *–interleave=<lista_de_nós>*. A figura 14 mostra a distinção entre uma política de alocação comum e a política de alocação intercalada que pode ser realizada com o uso da ferramenta *numactl*:

(a) Alocação não intercalada

(b) Alocação intercalada

Figura 14.: Duas possíveis abordagens de alocações de páginas de memória em arquiteturas NUMA com 2 bancos de memória.

O tamanho dos segmentos de memória pode ser também configurado com a opção *–length* em KB (k), MB (m) ou GB (g). Se não for especificada nenhuma unidade com o valor, este será interpretado como um número de bytes. Este tipo de funcionalidades ao nível da manipulação dos dados de memória acrescenta um novo valor a este estudo porque esta distribuição de páginas pode ser configurada de forma a distribuir os dados pelos diferentes bancos de memória da arquitetura NUMA.

Esta ferramenta pode ser também utilizada para compreender melhor a organização do hardware da arquitetura NUMA mostrando informação da tabela de localidade SLIT (*System Locality Information*) utilizando a opção *–hardware* como apresentado em seguida:

```
[a59905@compute-641-10 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 32733 MB
node 0 free: 31537 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 32768 MB
node 1 free: 31219 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

O comando foi executado num nó computacional do cluster SeARCH[4] com 2 processadores numa configuração NUMA. Esta arquitetura é composta por dois *Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz* da família *Ivy-Bridge*, cada um com 16 núcleos (*Hyper-Threading* incluído) e um banco de memória local de 32GB de RAM em cada processador.

numastat

O *numastat* é também uma ferramenta para Linux capaz de mostrar informação acerca das alocações de memória por cada nó NUMA. Esta ferramenta pode ser vista como uma ferramenta de monitorização que recolhe eventos relacionados com o número de alocações nos nós de uma arquitetura NUMA. A monitorização pode ser realizada em intervalos de tempo definidos com o uso do comando *watch* por forma a atualizar o número de *hits* e *misses* de alocações em cada nó NUMA. A ferramenta retorna os seguintes eventos:

- **numa_hit**: Refere-se ao número de páginas destinadas a serem alocadas num determinado nó e cuja alocação nesse mesmo nó foi bem sucedida;
- **numa_miss**: Mostra o número de alocações que se tentou realizar no presente nó, mas que foram alocadas noutro;

4 SeARCH é um cluster criado para serviços e investigação localizada na Universidade do Minho.

- **numa_foreign**: Corresponde ao número de alocações de páginas que tentaram ser alocadas num nó vizinho mas que acabaram por ser alocadas no presente nó;
- **interleave_hit**: Representa o número de alocações que foram intercaladas em cada nó com sucesso;
- **local_node**: Funciona como um contador que é incrementado quando um processo que corre num determinado nó, realiza alocações no mesmo nó;
- **other_node**: Representa um contador similar ao local_node mas que conta as alocações feitas num outro nó.

De acordo com os manuais do Linux sobre ferramentas, os eventos NUMA deste tipo apenas estão disponíveis apartir de versões do *Kernel* posteriores à versão 2.6.

migratepages

A ferramenta *migratepages*, dado o PID de um processo, permite migrar as páginas desse processo de um nó NUMA para outro. A invocação da ferramenta segue a sintaxe:

migratepages <*pid*> <*nó_origem*> <*nó_destino*>

Quando invocada a ferramenta irá migrar a localização física das páginas de memória do processo correspondente sem quaisquer alterações no espaço virtual de endereçamento. Esta ferramenta pode ser utilizada para minimizar a distância entre a localização a que os processos se encontram das páginas de memória a que acedem[26]. A utilização desta ferramenta permite, num nível de abstração superior, melhorar a localidade das páginas de memória em relação à localização dos processos que acedem a essas páginas.

3.4 CHAMADAS AO SISTEMA

Existem algumas chamadas a sistema disponíveis ao nível do *kernel* que podem ser usadas para controlar as alocações de páginas de memória entre nós NUMA e fazer *binding* de fios de execução a processadores específicos. As definições destas chamadas ao sistema estão presentes na biblioteca *numaif* e requerem um *kernel* NUMA-aware.

As chamadas a sistema desta biblioteca mais relevantes para manipulação de processos e zonas de memória são:

1. **mbind**: Permite configurar uma nova política de alocação de memória definida pelo programador;
2. **migrate_pages**: Pode ser utilizado para mover páginas de memória dado um determinado PID de um nó NUMA para um novo nó;
3. **move_pages**: É similar à função *migrate_pages* mas em vez de mover todas as páginas de memória associadas a um determinado PID, apenas move páginas especificas sendo necessário passar um apontador para o array de páginas que devem ser movidas como argumento à função juntamente com um array de inteiros que identificam os nós de destino de cada página;
4. **set_mempolicy**: Configura uma nova política de alocação de memória para o processo que invocou a função;
5. **get_mempolicy**: Ao contrário da chamada *set_mempolicy*, esta retorna a política de memória definida para o processo que invocou a função. Pode também ser utilizada para voltar à politica definida por omissão que foi anteriormente alterada com uso da chamada *set_mempolicy*.

3.5 OTIMIZAÇÃO AO NÍVEL DA PROGRAMAÇÃO

As aplicações em C/C++ podem utilizar *malloc* como alocador de memória. Ao utilizar o *malloc*, é retornado um apontador para a região de memória apesar do *kernel* ainda não ter criado as páginas de memória. A criação das páginas na memória apenas irá ocorrer quando o fio de execução num determinado núcleo *toca* nessa região de memória sendo que essa memória será então alocada na DRAM do nó NUMA a que o núcleo pertence. O *toque* que leva à criação das páginas na memória ocorre, por exemplo, quando um fio de execução inicializa uma posição de um vetor com um determinado valor.

Considere-se a criação de um vetor em C no nó 0 de uma arquitetura NUMA:

```
int *a = (int*) malloc(N*sizeof(int));

for (i=0; i<N; i++)
  a[i] = 0;                    /* Inicializacao sequencial */
#pragma omp parallel for
{
    for (i=0; i<N; i++)
    .... computacao realizada com a[i] em paralelo ....
}
```

Neste primeiro exemplo, se o primeiro fio de execução que toca no vetor ***a*** após a realização do *malloc* pertencer a um núcleo no processador 0, todas as páginas de memória associadas a este vetor serão armazenadas na memória local do processador 0. Os fios de execução que irão realizar computações com o vetor ***a*** dentro da região paralela podem correr em processadores de nós NUMA vizinhos originando acessos remotos.

A técnica designada por *First Touch*, envolve a realização da inicialização em paralelo por múltiplos fios de execução:

```
int *a = (int*) malloc(N*sizeof(int));

#pragma omp parallel for
for (i=0; i<N; i++){
        a[i] = 0;              /* Inicializacao em paralelo */
    .... computacao realizada com a[i] em paralelo ...
}
```

Com esta (simples) alteração, as posições do vetor que sejam inicializadas pelos fios de execução do processador 0, terão as respetivas páginas de memória alocadas na memória do processador 0, e as posições inicializadas pelos fios de execução no processador 1 terão as respetivas páginas de memória alocadas na memória do processador 1.

As alocações em bancos de memória distintos fazem com que os fios de execução que corram em cada um dos processadores acedam localmente à memória alocada evitando tráfego nos conetores. Com a aplicação desta técnica, a alocação das páginas de memória do vetor é dividida pelos bancos de memória dos 2 (ou mais) nós evitando que sejam

todas alocadas no mesmo nó sequencialmente, como acontece na primeira codificação. Esta divisão permite o aumento da largura de banda sendo que se espera usar o dobro da largura de banda numa arquitetura 2-socket que se refletem em ganhos de desempenho na aplicação. Numa arquitetura NUMA 4-socket (4 nós) a alocação pode ser dividida por 4 processadores em paralelo o que pressupõe um aumento de 4 vezes no uso da largura de banda.

A adoção desta técnica faz com que implicitamente se consiga diminuir a distância a que os fios de execução se encontram dos dados de memória. Esta técnica será utilizada nos casos de estudo utilizados na componente experimental deste trabalho na secção 4.

3.6 MECANISMOS DA JAVA VIRTUAL MACHINE

À medida que a linguagem Java foi evoluindo, foram introduzidas otimizações na JVM que permitem aumentar o desempenho dos códigos em arquiteturas mais recentes. No que toca às arquiteturas NUMA, a JVM das versões do Java mais recentes possuem opções que ativam optimizações por forma a aumentar o seu desempenho em arquiteturas NUMA. A Oracle colocou as primeiras melhorias de desempenho relacionadas com NUMA no Java HotSpot™ Virtual Machine com a versão *SE 6u2*[31]. Esta novas melhorias envolvem uma extensão ao *Parallel Scavenge Garbage Collector* e a adição de um novo alocador *NUMA aware* que muda a forma como são realizadas as alocações de memória entre nós NUMA[31].

3.6.1 *Alocador NUMA aware da JVM*

Em Java, novos objectos são criados no *eden space* da *young generation* da heap. Esta região passou a ser controlada pelo alocador NUMA aware que separa o espaço em múltiplas zonas que serão alocadas no banco de memória de cada um dos nós[31]. O alocador procura manter os objetos próximos dos fios de execução que realizam a sua alocação, assumindo que será esse fio de execução que realmente irá usar o objeto. As regiões de memória criadas pelo alocador não são estáticas e podem ser redimensionadas dinamicamente por forma a aumentar a taxa das alocações realizadas por fios de execução a correr em diferentes nós[31]. O alocador tenta também diminuir as as-

simetrias de latência com que os fios de execução acedem à *young*, *old*, e *permanent generations*[31].

O alocador NUMA aware em Java pode ser habilitado com a opção *-XX:+UseNUMA* e, por recomendação da Oracle deverá ser utilizado em conjunto com o *Parallel Scavenge garbage collector*. Este é o *garbage collector* incluído por omissão na Java HotSpot Virtual Machine e que pode ser ativado explicitamente com a opção *-XX:+UseParallelGC*. No *SPEC JBB 2005* a Oracle anunciou um aumento de desempenho à volta dos 30% numa arquitetura Opteron da AMD de 32 bits, e 40% de aumento de desempenho em 64 bits com 8 processadores com o uso da opção *-XX:+UseNUMA*[31].

Além do alocador NUMA aware, a JVM disponibiliza ainda um conjunto de outras opções relacionadas com NUMA abordadas em seguida.

3.6.2 *Optimizações NUMA na JVM*

A forma atual de ativar otimizações NUMA na JVM (nas versões 8 e 9 do Java) é ativando certas opções que mudam o comportamento da JVM na forma como a memória é gerida. A tabela seguinte mostra a lista de otimizações que podem ser ativadas com adição da *flag* seguindo a sintaxe **-XX:+<opção>**:

Opção	Tipo	Valor (omissão)	Descrição
UseNUMA	Boolean	False	Introduzido para habilitar o alocador de memória NUMA aware. É uma *flag* está disponivel não está ativada por omissão.
UseParallelGC	Boolean	False	Ativa um garbage collector paralelo designado por Parallel Scavenge
UseParallelOldGC	Boolean	False	Utiliza o garbage collector paralelo antigo.
ForceNUMA	Boolean	False	Não existe muita informação acerca do tipo de optimizações que são introduzidas.
UseAdaptiveNUMAChunkSizing	Boolean	True	Habilita a JVM a adaptar dinamicamente o tamanho dos *chunks* escalonados nos nós NUMA.
NUMAChunkResizeWeight	Integer	20	Utiliza uma percentagem (entre 0 e 100) para pesar o tamanho do *chunk* atual ao longo da computação.
NUMAPageScanRate	Integer	256	Define o número máximo de páginas por procedimento
NUMASpaceResizeRate	Integer	1G	Define a quantidade máxima de espaço que pode ser realocado por colecção.
NUMAStats	Boolean	False	Imprime informação NUMA detalhada sobre a heap.

3.7 DETECÇÃO DE ALVOS DE OTIMIZAÇÃO

As ferramentas de *profilling* podem ser seguidas para verificar se o desempenho de uma aplicação está a ser afetado pelo padrão de acessos à memória numa arquitetura NUMA. Estas ferramentas permitem recolher um conjunto de eventos relacionados com o número de acessos locais, acessos remotos, medição da largura de banda por nó, medição do tráfego nos conectores que ligam os processadores entre outros. O número de acessos locais e remotos podem ser obtidos com recurso a contadores embutidos no hardware encarregues de contabilizar esses acessos no período de execução da aplicação. Algumas ferramentas (por exemplo PAPI em C), fornecem abstrações que permitem o acesso à leitura destes contadores através de instrumentação do código definindo as regiões a monitorizar.

Contudo, dada a necessidade de instrumentação, para Java este suporte é mais reduzido mas existem algumas bibliotecas. O PAPIJ por exemplo, permite o uso de PAPI para Java através do Java Native Access (JNI).

Existe ainda a ferramenta de *profilling* Intel VTune com suporte para Java, e pode ser utilizada para consultar os contadores de memória.

Existem outras ferramentas para medir o número de acessos não baseadas em instrumentação mas por amostragem como por exemplo o *perf* (disponibilizada pelo kernel[5] Linux). Esta ferramenta além de possuir suporte para aplicações em Java, permite medir diversos eventos de hardware que ajudam a compreender se uma aplicação têm um número elevado de *cache misses* nos diferentes níveis, o número de ciclos gastos, e alguns contadores de memória que permitem quantificar o número de acessos locais e remotos. Desta forma, com estes contadores é possível confirmar o nível de sucesso de determinada afinidade de fios de execução e respetivas alocações de memória realizadas com os mecanismos estudados até agora. Por exemplo, se for utilizada uma política de afinidade que resulte num elevado número de acessos remotos registado, e baixo número de acessos locais, estas ferramentas permitem perceber essa política de afinidade deve ser substituída por outra com melhores indicadores. No entanto, existem algumas desvantagens associadas à utilização de contadores nas ferramentas de *profilling*. Uma vez que os contadores são fornecidos pelo hardware e embutidos na própria lógica dos

5 A partir das versões 2.6.31 do kernel [23]

processadores e/ou memória, o suporte para determinados contadores tem que ser assegurado pelo próprio hardware. A utilização destas ferramentas utiliza normalmente uma camada intermédia responsável pela tradução do código do contador no hardware por um nome que caracteriza o evento que esse contador mede, de forma mais compreensível para o utilizador. É contudo frequente o hardware suportar diversos contadores que não são diretamente traduzidos pela ferramenta de acesso aos contadores. Isto acontece sobretudo com alguns contadores nativos e pode acontecer igualmente com contadores de memória que contabilizam acessos locais e remotos sendo necessário algum esforço adicional para os habilitar. Outra das desvantagens associadas a estas ferramentas está no facto dos valores medidos por vezes serem difíceis de justificar e de certos contadores nem sempre serem precisos.

Existem outros tipos de ferramentas que permitem saber se um dos alvos de otimização está relacionado com as alocações de memória. Se os bancos de memória não estiverem a ser utilizados eficientemente é possível detetar bancos de memória que registam baixa largura de banda. Isto pode ocorrer por exemplo, quando os fios de execução não estão distribuidos pelos processadores a utilizarem os seus bancos de memória locais, ou caso todos os fios de execução estejam a usar sempre o mesmo banco de memória. É comum utilizar-se o STREAM benchmark[6] para medição da largura de banda da memória de um sistema de computação. Contudo a utilização deste é limitada a programas em C e Fortran não sendo o mais indicado para medir a largura de banda de memória numa arquitetura NUMA. Além do STREAM foi também considerada uma outra ferramenta de medição de largura de banda: O *Processor Counter Monitor (PCM)*[7]. A ferramenta PCM-Memory é uma das ferramentas incluídas no *PCM* que além de ser compatível com aplicações em Java, tem a vantagem adicional de ser mais indicada para arquiteturas NUMA dado o seu suporte de medição de largura de banda por nó durante a execução. A largura de banda medida possui uma granularidade consideravelmente fina mostrando concretamente a largura de banda por canal de memória, separada por largura de banda de leituras, escritas e o total por DIMM. Além do *PCM-Memory* o pacote inclui ainda uma ferramenta, *PCM-NUMA* que permite perceber a quantidade de tráfego realizado pelo QPI por forma a perceber se a perda de desempenho está relacionada com o excesso de tráfego no conetor.

6 Disponivel em: https://www.cs.virginia.edu/stream/
7 Repositório da ferramenta: https://github.com/opcm/pcm

4

COMPONENTE EXPERIMENTAL

O capítulo anterior apresentou diferentes abordagens para melhorar o desempenho de aplicações em arquiteturas NUMA desenvolvidas em linguagens como o Java e C. O trabalho experimental que se segue neste capítulo conta com uma abordagem exploratória onde diversas configurações de afinidade são testadas por forma a identificar as melhorias no desempenho de dois casos de estudo. Neste capítulo são também exploradas alterações no código dos casos de estudo visando aumentar o desempenho em arquiteturas NUMA.

Os resultados experimentais são apresentados numa perspetiva de comparação entre os ganhos de desempenho conseguidos com diferentes testes, para diferentes configurações de afinidade por forma a encontrar as configurações de afinidade mais adequadas para cada versão. Das várias ferramentas que foram estudadas para validação dos resultados, foram utilizadas duas ferramentas de profilling baseados em contadores do PMU (*Performance Monitor Unit*) e análise de largura de banda por nó. A ferramenta *PCM-NUMA* foi a ferramenta utilizada para recolher informação dos contadores de acessos locais e remotos, e a ferramenta PCM-Memory para analisar a largura de banda utilizada.

4.1 CASOS DE ESTUDO

Neste estudo são explorados dois casos de estudo: *Successive over relaxation (SOR)* e fatorização LU (LUFACT). Ambos os algoritmos envolvem a resolução de sistemas de equações lineares que pertencem a uma categoria de algoritmos que se caracteriza por serem exigentes em termos de largura de banda. O estudo é feito com base na análise de desempenho deste algoritmos numa arquitetura NUMA de implementações sequenciais e paralelas de ambos nas linguagens Java e C.

Os testes foram realizados com diferentes tamanhos de matrizes. As implementações paralelas em C utilizam OpenMP e para as implementações em Java, foi usada a classe *Thread* nativa. As implementações em Java correspondem a versões otimizadas com recurso à biblioteca *AOmpLIB*[28].

4.1.1 *Successive Over Relaxation (SOR)*

Este algoritmo é uma alternativa ao método *Gauss-Seidel* para resolução de equações diferenciais parciais. É um algoritmo conhecido na álgebra linear por ser um algoritmo de rápida convergência em comparação outros métodos iterativos. A matriz envolvida é iterada um determinado número de vezes. Cada elemento na nova matriz que é calculado utilizando quatro elementos vizinhos [36]. O paralelismo deste algoritmo utiliza a variante *red-black* do algoritmo seguindo uma estratégia *heartbeat* particionando a matriz original em múltiplas secções. Estas secções são atribuídas a diferentes fios de execução para serem processadas em paralelo sendo que o trabalho de cada fio de execução envolve a execução de um determinado número de iterações. Por cada iteração é necessário realizar sincronização entre fios de execução. Esta implementação é composta por três classes: *JGFSORBenchSizeA*, *SOR* e *JGFSORBench*.

A classe *JGFSORBenchSizeA* contém o método *main* e três variáveis: o tamanho da matriz envolvida no sistema de equações, o número de fios de execução a serem criados, e um *booleano* para ativar a validação da solução final. A segunda classe *SOR* contém as definições dos métodos *SORrun* e *SORRunner* responsáveis pela criação dos fios de execução, e que implementa o particionamento da matriz entre os fios de execução

respetivamente. A terceira classe extende a classe *SOR* e é usada para reservar espaço para a matriz, preenchê-la com valores aleatórios e correr o *kernel* do algoritmo.

Esta implementação corresponde a uma versão otimizada do *SOR* da *Java Grande Forum Benchmark Suite* cujo número de instruções condicionais dentro do *kernel* do algoritmo foi reduzido para aumento de desempenho da versão sequencial.

4.1.2 *Fatorização LU (LUFACT)*

Este caso de estudo resolve um sistema de equações lineares usando a conhecida decomposição LU (Lower Upper") introduzida por Tadeusz Banachiewicz em 1938 [35]. O algoritmo decompõe a matriz original do sistema de equações num produto de matrizes triangulares seguido de resolução por substituíção inversa. Á semelhança do algoritmo SOR, a implementação em Java do LUFACT pertence igualmente à coleção de benchmarks *Java Grande Forum Benchmark Suite*.

A nível de estrutura, a aplicação tem uma organização semelhante à estrutura do SOR. Existe uma classe adicional designada por *ThreadTeam* que contém a criação das tarefas, fios de execução e barreiras. A classe *JGFLUFactBenchSizeA* contém o método *main* com as mesmas três variáveis como no SOR: o tamanho da matriz, o número de fios de execução, e um valor booleano para validação. A classe *Linpack* contém as definições de métodos envolvidos na geração da matriz, fatorização e resolução tais como: *matgen*, *dgefa*, e *rowElimination*. São utilizados também outros métodos auxiliares da *LAPACK* como *daxpy*, *dscal*, *idamax* entre outros. A classe *JGFLUFactBench* extende a classe *Linpack* e contém métodos de configuração da matriz, inicialização, criação da equipa de fios de execução, e execução do *kernel* do algoritmo (i.e o método *dgefa* para a fase de factorização e *dgesl* para a fase de resolução). A fase de fatorização corresponde à parte computacionalmente mais exigente deste algoritmo[28]. O paralelismo ocorre no método de eliminação da linha dentro do método *dgefa* responsável por computar a fatorização.

4.2 AMBIENTE DE TESTES E EXPERIMENTAÇÃO

Os testes foram realizados com um nó do cluster SeARCH da Universidade do Minho. Este cluster foi criado para serviços e investigação em computação avançada disponível para a comunidade académica oferecendo um ambiente heterogéneo de microarquiteturas [1].

O nó computacional utilizado neste estudo possui dois Intel® Xeon® E5-2650v2 da micro-arquitetura *Ivy Bridge* com as especificações referidas na tabela 1:

Nome do nó	compute-641
Processador	Intel Xeon E5-2650v2
# Processadores	2
Micro-arquitetura	Ivy Bridge
# Núcleos por processador	8
# Núcleos virtuais por processador	16
Frequência de relógio	2.6 - 3.4 GHz (Max.)
Cache Nível 1 (L1)	64KB por núcleo
Cache Nível 2 (L2)	256KB por núcleo
Cache Nível 3 (L3 / LLC)	20MB *SmartCache* (partilhada)
Conetor	QPI a 8 GT/s $\sim$ 16GB/s (unidirecional)
# Ligações por conetor	2
Largura de banda total do conetor	32 GB/s (bi-direcional)
# Canais de memória / nó NUMA	4
Memória RAM total / nó NUMA	32GB
Largura de banda anunciada	59.7 GB/s

Tabela 1.: Nó computacional do cluster SeARCH utilizado na componente experimental

Este nó computacional possui dois processadores multinúcleo com memória local formando uma arquitetura *2-socket* NUMA. A replicação de testes em arquiteturas NUMA *4-socket* ou *8-socket* não foi realizada neste estudo uma vez que não existem nós computacionais com tais arquiteturas neste *cluster*. As especificações deste nó computacional está presente na figura 15 obtida com a ferramenta *lstopo*:

1 Mais informação sobre o *SeARCH* disponível em: http://search.di.uminho.pt/wordpress/

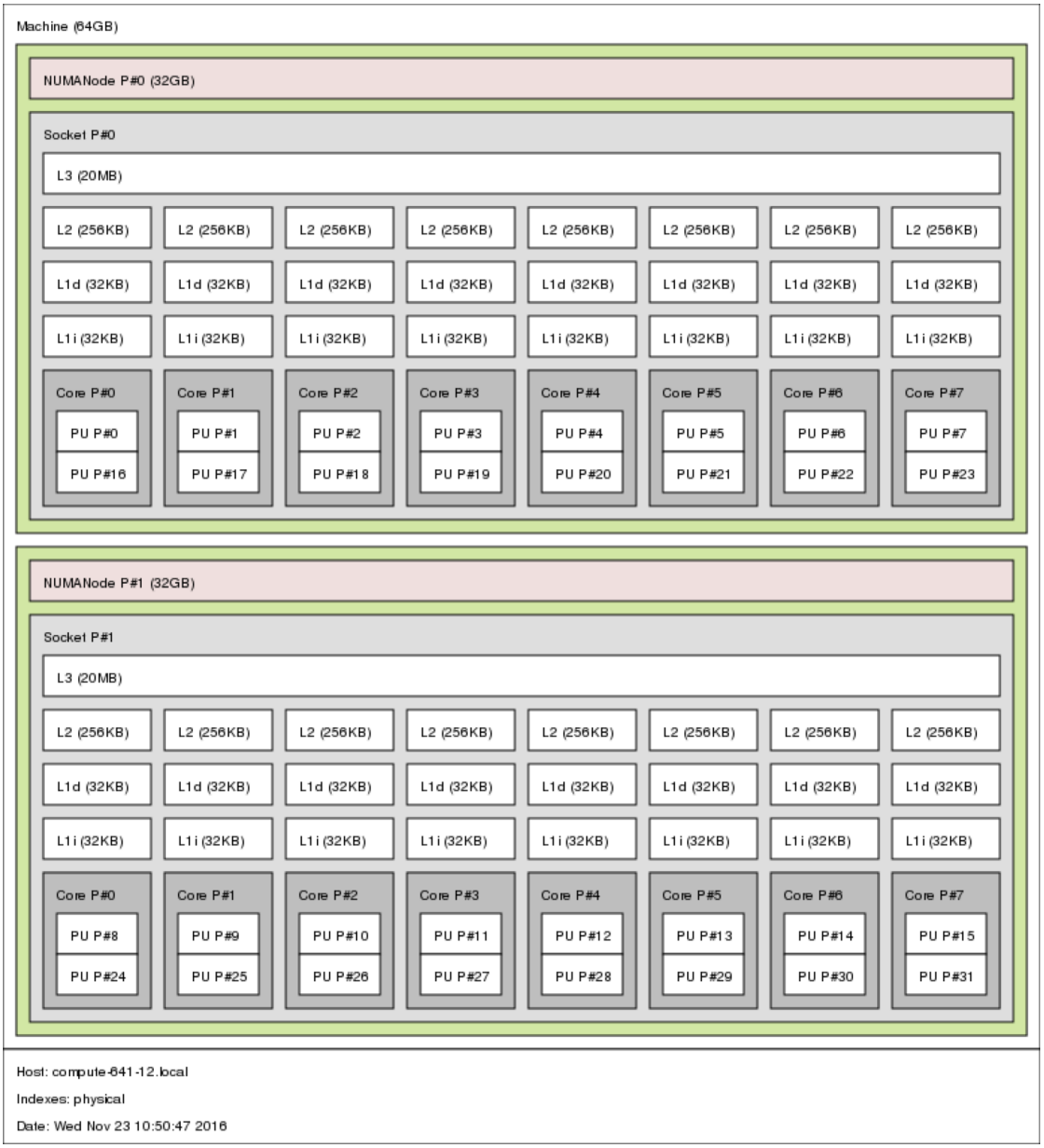


Figura 15.: Arquitetura NUMA do nó compute-641

Por observação da imagem é possível verificar um total de 32 núcleos de processamento com uma hierarquia de memória composta por três *níveis* de cache e dois bancos de memória principal (DRAM). O nível de cache L1 é dividido em cache de instruções e dados por núcleo cada um com 32 KB o que representa um total de 64 KB por núcleo

e um total de 1024 KB nos dois nós NUMA. Existe um total de 4096 KB de memória L2 e 40 MB de cache L3 partilhada entre todos os núcleos.

Os testes foram realizados com dois tamanhos de matrizes distintos: um tamanho cujos dados cabem na cache L3 e outro armazenado em DRAM. Os detalhes em relação aos tamanhos das matrizes estão disponíveis no Anexo A. Nesta secção apenas são apresentados os resultados para dois tamanhos diferentes, uma vez que os resultados são semelhantes para os outros tamanhos.

A interligação dos dois processadores é assegurada pelo conetor QPI com duas ligações com um total de cerca de 32 GB/s de largura de banda enquanto que cada banco de 32 GB de RAM tem uma largura de banda anunciada de 59.7 GB/s. Como constatado anteriormente na secção 2.3.7, este conetor é uma limitação na arquitetura NUMA pois existe uma diferença de quase 50% entre a quantidade de dados que pode ser transferida por segundo através do QPI e a largura de banda dos acessos locais. A inspeção da arquitetura NUMA presente num sistema de computação pode ser também consultada através da *System Locality Information (tabela SLIT)* como mostrado na secção 3.3.4.

Os testes realizados utilizam o tempo de execução do *kernel* dos algoritmos como principal métrica de desempenho. Todos os testes foram realizados com reserva de todo o nó computacional por forma a assegurar o mínimo de carga adicional originada por outras aplicações ou serviços. Como apresentado na tabela 1, o nó computacional suporta uma frequência variável entre 2.6 e os 3.4 GHz. A realização dos testes com uma frequência variável altera as condições de experimentação em cada execução o que pode conduzir a resultados pouco precisos. Deste modo, sempre que possível foram utilizados nós com uma frequência fixa nos 2.6 GHz excluindo assim os efeitos da tecnologia *Intel® Turbo Boost*. Os tempos de execução de cada teste foram obtidos com recurso a 10 amostras de onde foi extraída a mediana desses tempos com três casas decimais.

As implementações em C dos algoritmos utilizados foram compiladas com nível de optimização -O3 do compilador GCC 5.3.0 e ICC 16. Os testes Java utilizam a Oracle JDK do Java 8 (versão 1.8.0_20).

4.3 RESULTADOS EXPERIMENTAIS: INTRODUÇÃO

Como foi constatado ao longo do capítulo 3, a afinidade de fios de execução e as políticas de alocação de memória são duas abordagens para aumento do desempenho das aplicações em arquiteturas NUMA. Dependendo da linguagem de programação adotada existem diferentes mecanismos suportados para explorar afinidades e alocações de memória. Nesta componente experimental são explorados um conjunto de testes cujo objetivo é perceber quais as estratégia de afinidade e de alocação de memória para aplicações Java em arquiteturas NUMA que proporcionam o melhor desempenho. Os casos de estudo explorados nos testes correspondem às versões sequenciais e versões paralelas em memória partilhada dos algoritmos SOR e LUFACT em Java, C e implementações *First Touch*.

4.4 CARACTERIZAÇÃO DOS TESTES EM JAVA

No primeiro grupo de testes foi considerada a execução original (sequencial e versão de memória partilhada) do algoritmo sem quaisquer otimizações. Realizou-se um segundo teste baseado no código do programa original (sem alteração do código) ao qual é adicionada opção *-XX:+UseNUMA* e *-XX:+UseParallelGC* que ativam o alocador NUMA aware do Java e o *Parallel Scavenge Garbage Collector*. A ativação deste coletor de lixo em simultâneo com a ativação do alocador *NUMA aware* foi feita seguindo a recomendação da Oracle[31]. No entanto, a ativação do *Parallel Scavenge Garbage Collector* não apresentou diferenças significativas no desempenho, pelo que os ganhos conseguidos com este teste refletem essencialmente os ganhos conseguidos com a ativação do alocador.

O terceiro grupo de testes utiliza a ferramenta *numactl*. Esta ferramenta externa pode ser utilizada com aplicações Java sem necessidade de alterar o código fonte e permite controlar os processadores que devem ser utilizados para processar os fios de execução das aplicações paralelas em Java. Apesar da ferramenta oferecer alguma flexibilidade e suporte em relação às alocações de memória sobre os processadores, não oferece um nível de granularidade que garanta controlo restrito aos fios de execução criados pela

aplicação. Como consequência, os fios de execução da aplicação e aqueles que decorrem do funcionamento da JVM são tratados de forma indiferenciada.

A ferramenta numactl disponibiliza diferentes políticas de alocação de memória. É possível instruir o *kernel* a realizar as alocações preferencialmente num determinado nó (modo *preferred*), realizar todas as alocações apenas no nó em que a aplicação começou a executar (com opção *localalloc*) ou ainda utilizando uma política de distribuição intercalada (*interleave*). Nesta última, as alocações são realizadas pelos nós especificados seguindo uma política *round-robin* para distribuir as páginas pelos nós. As alocações podem também ser realizadas de forma explícita com recurso à opção *membind* onde o programador pode decidir os bancos de memória a serem utilizados para as alocações à semelhança do que acontece com a opção *cpubind* para restringir a execução em determinados processadores. O teste com as opções *–cpubind=0 –membind=0* limitam a execução aos núcleos do processador 0 e as alocações de páginas de memória são feitas no nó desse processador. Este teste maximiza o número de acessos locais evitando acessos remotos uma vez que as alocações de memória são realizadas na memória do processador 0. Com este teste pretende-se perceber se a execução original é afetada a nível de desempenho por eventuais acessos remotos. Os testes com as opções *–coubind=0 –membind=1*, maximiza o número de acessos remotos. Este teste foi feito para perceber o custo associado com acessos remotos. Também foi realizado um teste com a utilização combinada do alocador *NUMA aware* do Java e com a ferramenta *numactl*.

4.5 IMPACTO DAS AFINIDADES EM JAVA NO DESEMPENHO

Os resultados nas figuras 16, e 17 referem-se aos testes realizados para os algoritmos SOR e LUFACT em Java cujos tamanhos de dados cabem nas *caches* L3 disponiveis. Os ganho de desempenho das figuras são relativos à versão sequencial da execução original (linhas a cinzento).

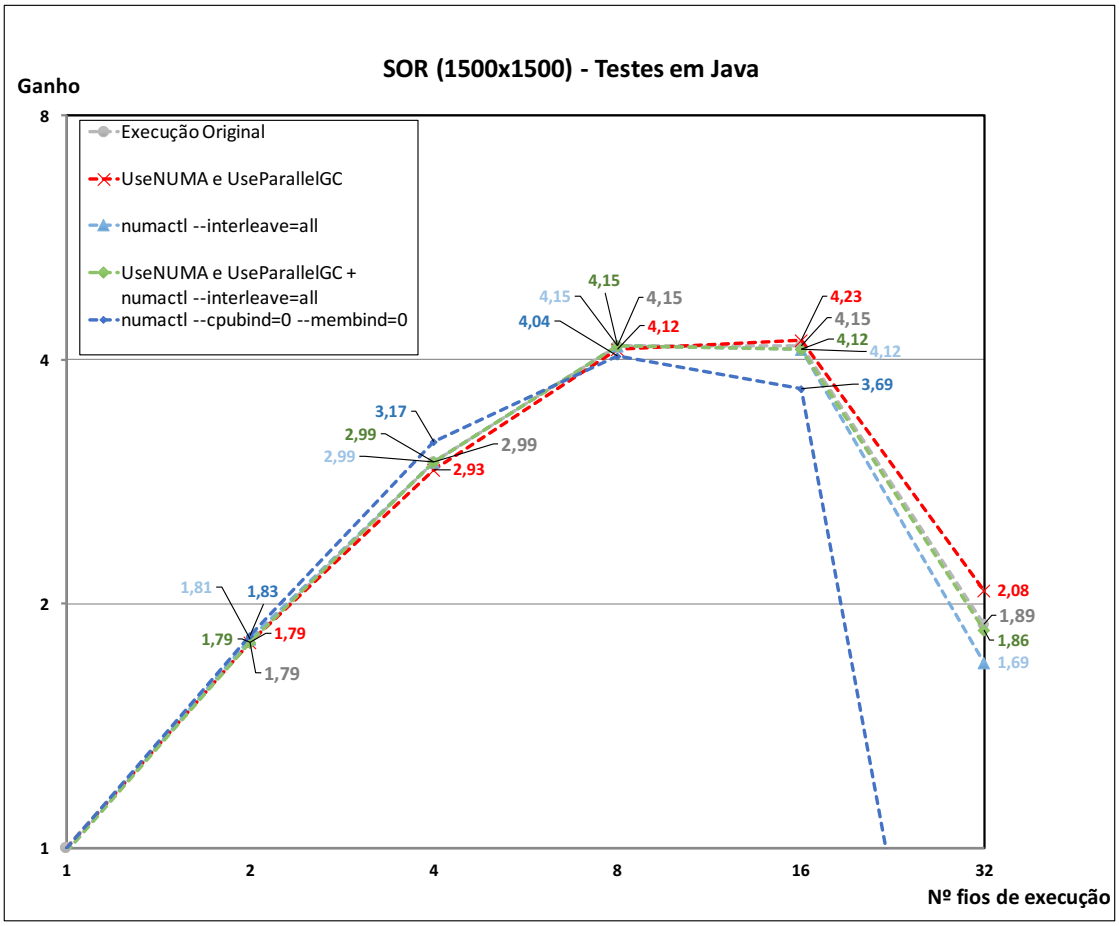


Figura 16.: SOR Java: ganhos desempenho com afinidades para dados em L3 (1500x1500).

Os testes revelam que o algoritmo SOR original escala até ao máximo de 8 fios de execução que corresponde ao número de fios de execução que esgotam os núcleos de um dos processadores. Manter todos os fios de execução no mesmo processador (linha azul escura) compensou até aos 4 fios de execução. Constata-se que apartir de 8 fios

de execução, começa a existir contenção, e deixa de fazer sentido manter os fios de execução no mesmo processador em que estão alocados os dados na memória. O uso do alocador *NUMA aware* na JVM fez com que algoritmo escalasse um pouco mais que a versão original sendo que essa diferença não é significativa.

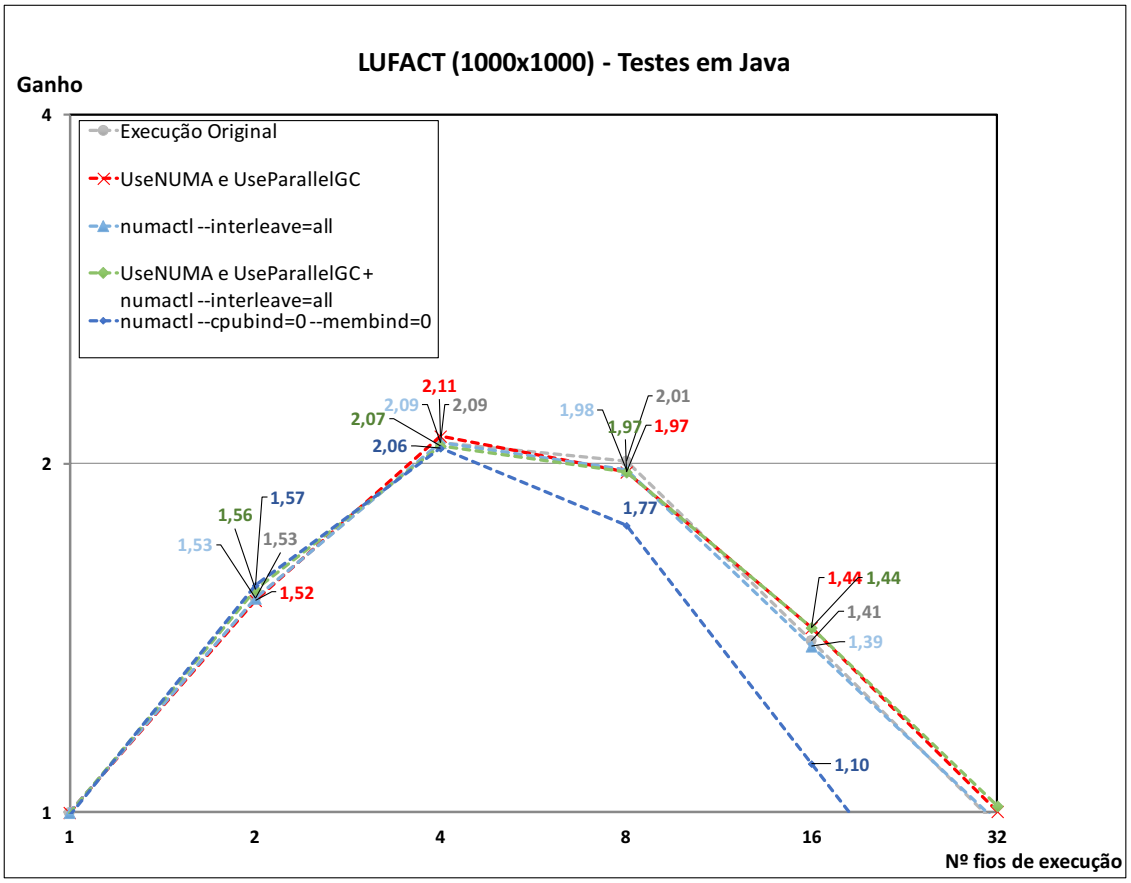


Figura 17.: LUFACT Java: ganhos desempenho com afinidades para dados em L3 (1000x1000).

Na figura 17 à semelhança do que acontece com o algoritmo SOR, também o LUFACT escala ligeiramente melhor com o alocador *NUMA aware* para este tamanho de dados. Nos testes para o algoritmo LUFACT em Java, percebe-se que a perda de desempenho com acessos remotos é na realidade bastante baixa, já que as curvas que maximizam acessos locais e remotos tem um comportamento semelhante à medida que se vai aumentando o número de fios de execução.

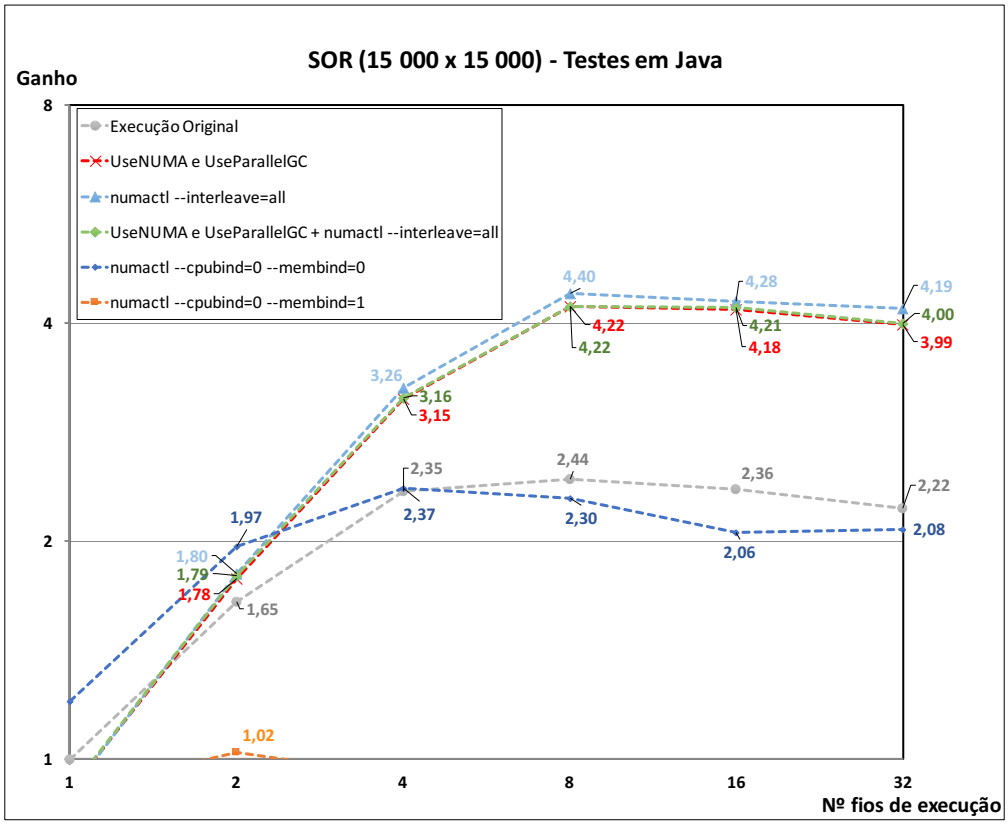


Figura 18.: SOR Java: ganhos desempenho com afinidades para o tamanho em DRAM (15000x15000).

Quando se aumentou para o tamanho de dados que cabe na DRAM é possível verificar um maior afastamento das curvas dos gráficos (figuras 18 e 19). Para este tamanho, verifica-se que ter os fios de execução restringidos ao primeiro processador a aceder a memória local (linha azul escura), compensa apenas para um ou dois fios de execução em comparação com os restantes. À semelhança do que se verificou para o tamanho em *cache* L3, também para o tamanho em DRAM a maximização dos acessos locais só compensa até aos 4 fios de execução nos dois algoritmos. Apartir desse limite, começa a verificar-se de forma mais expressiva os benefícios da utilização do alocador NUMA aware do Java, e o uso da ferramenta *numactl* com uma estratégia de intercalar páginas de memória pelos dois nós. A utilização destas abordagens permitiu tirar um maior partido da arquitetura NUMA. Em ambos os casos, a ferramenta *numactl* registou um ganho superior. A utilização combinada das duas abordagens (linha verde) não registou diferenças significativas a nível de ganho sendo que as curvas são coincidentes nos dois

algoritmos. Em resumo, no algoritmo SOR conseguiu-se um ganho máximo de 4.40 vezes com a ferramenta *numactl* quando o ganho máximo da execução original registou apenas 2.44 vezes. Para o algoritmo LUFACT, o ganho máximo de 4.22 foi conseguido igualmente com a ferramenta *numactl* face à execução original com um ganho de 2.71 vezes.

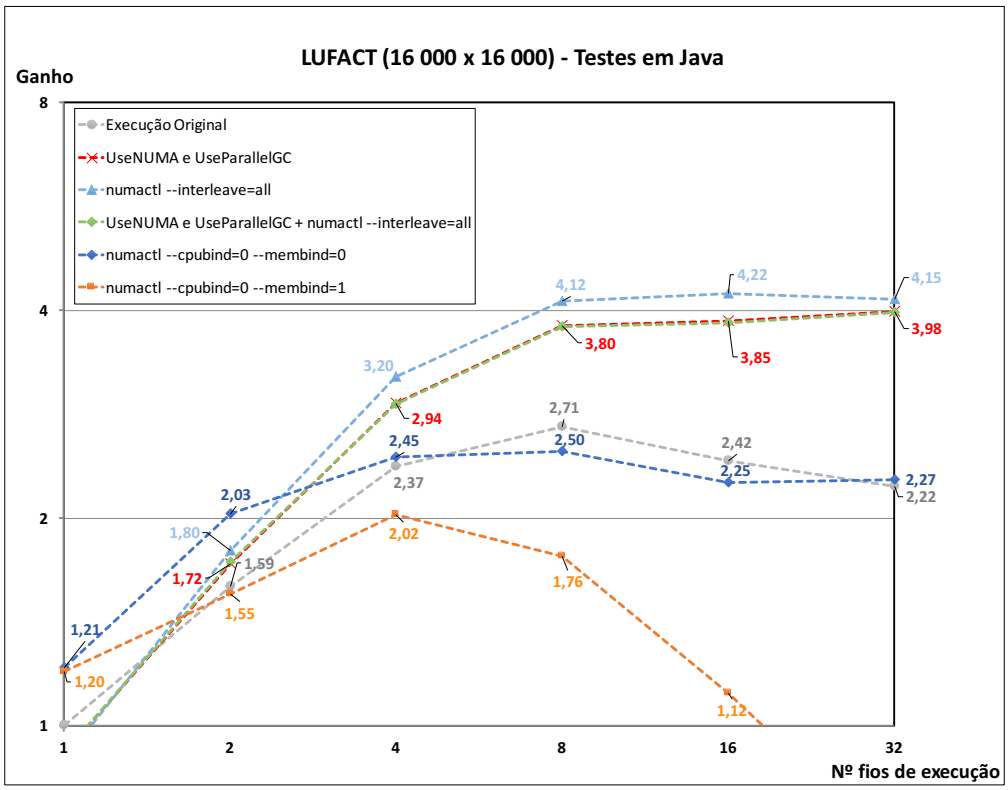


Figura 19.: LUFACT Java: ganhos desempenho com afinidades para o tamanho em DRAM (16000x16000).

4.6 ANÁLISE DA LARGURA DE BANDA E ACESSOS LOCAIS E REMOTOS

Uma análise sobre a largura de banda utilizada por cada nó NUMA, ajuda a perceber os ganhos de desempenho conseguidos com as diferentes configurações de afinidade. Existe uma relação direta entre os ganhos de desempenho atingidos com a largura de banda de memória. Os maiores valores de larguras de banda atingidos, estão relaciona-

dos com os testes de maior ganho em tempo de execução. Nas figuras 20 e 21 estão registadas as larguras de banda para os diferentes testes.

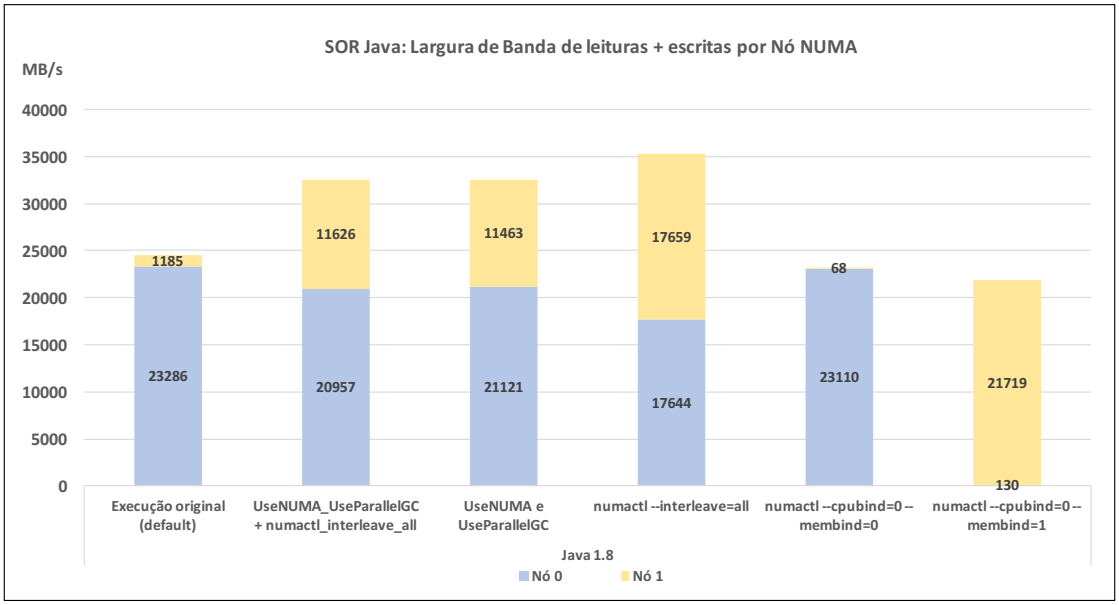


Figura 20.: SOR Java: Largura de banda para o tamanho em DRAM (15000x15000).

Verifica-se que a tendendência da JVM é por omissão realizar as alocações de memória num dos nós. Á medida que se aumenta o número de fios de execução, esta alocação, como constatado nos gráficos da secção 4.5, faz com que os acessos à DRAM limitem os ganhos.

Os testes sugeridos neste estudo demonstram que o aumento dos ganhos conseguidos com o uso de alocações de memória distintas das realizadas originalmente pela JVM, decorrem destas conseguirem atingir uma maior largura de banda ao distribuir os segmentos de memória de forma intercalada pelos dois nós NUMA. A ativação do alocador *NUMA aware* na JVM abordado na secção 3.6.1 com adição da opção *-XX:+UseNUMA*, aumenta a distribuição da largura de banda pelos dois nós NUMA. No entanto, para este teste no algoritmo SOR nota-se uma certa tendência da JVM a privilegiar um pouco mais a distribuição de memória no nó onde a aplicação iniciou a execução (realizando aproximadamente mais $2/3$ das alocações nesse nó). Pode-se concluir que a execução do SOR terá sido realizada no processador 0 com acessos de memória ao seu nó de memória local, uma vez que a execução original apresentou uma largura de banda nesse

nó, semelhante à obtida com o teste *–cpubind=0 –membind=0* do *numactl* no qual o ambiente de execução é limitado ao processador 0 e respectivo nó local.

O teste com *numactl* com política intercalada de páginas de memória *interleave*, revela um aumento de desempenho adicional ao conseguido com o alocador *NUMA aware* nos dois casos de estudo. O gráfico 18 revela que a curva azul clara correspondente à execução da ferramenta numactl está sempre ligeiramente acima da curva do teste *–XX:+UseNUMA* (a vermelho) pelo teste de largura de banda conclui-se que este ganho adicional é conseguido por uma divisão da largura de banda ainda mais equitativa pelos dois nós NUMA como se verifica na figura 20. Cerca de $1/2$ de largura de banda é garantida em cada um dos nós garantindo uma largura de banda total maior. Com o alocador NUMA aware, nem sempre se garante essa proporção como acontece com o algoritmo SOR (proporção de $2/3$).

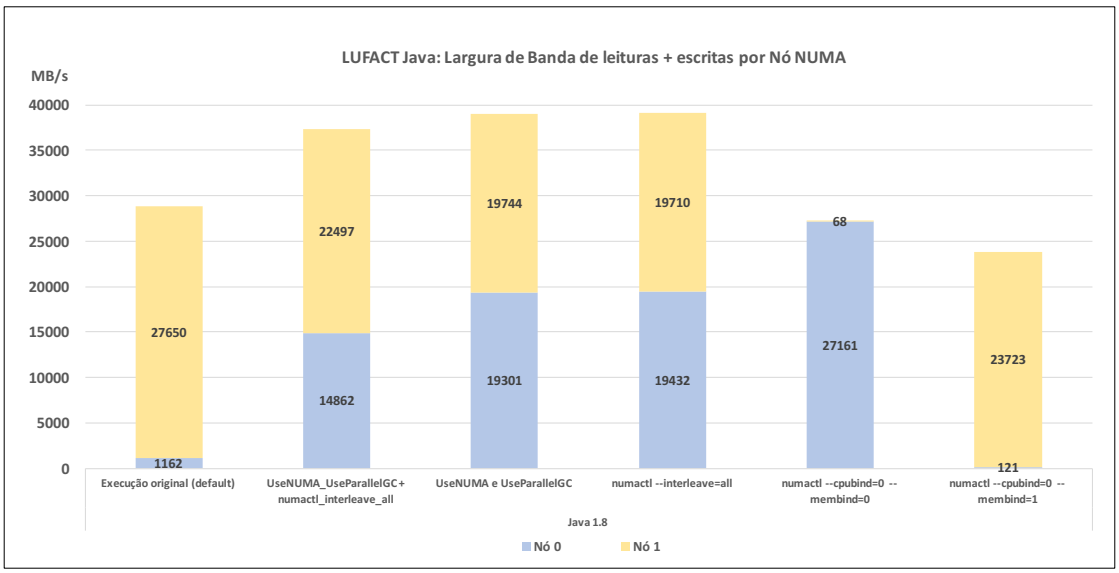


Figura 21.: LUFACT Java: Largura de banda para o tamanho em DRAM (16000x16000).

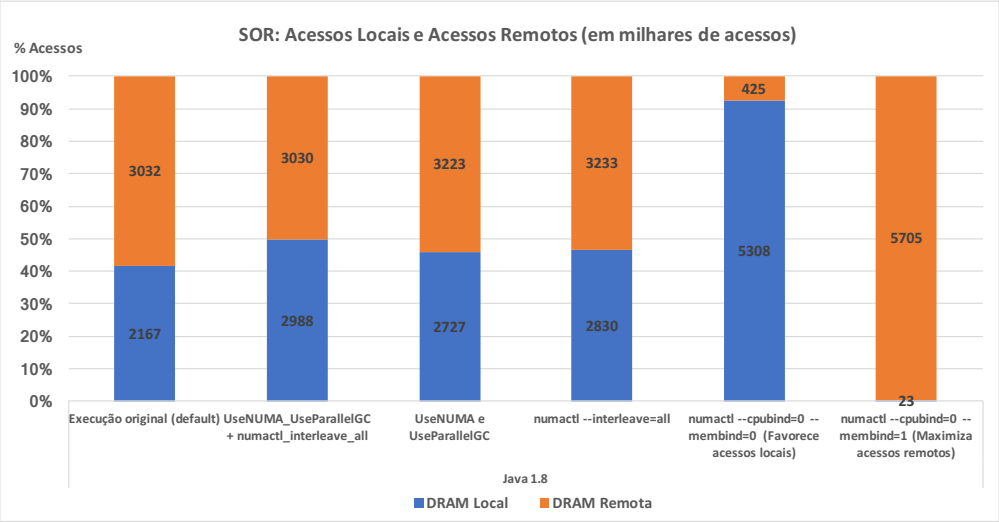


Figura 22.: SOR Java: Acessos Locais e Remotos (tamanho em DRAM - 15000x15000).

Os resultados dos contadores de memória nas figuras 22 e 23 permitem concluir que apesar de existir um número de acessos remotos mais elevado que o número de acessos locais em ambos os casos de estudo, todos os testes (à exceção dos testes em que favorecem acessos locais e remotos) apresentam uma proporção de quase 50% entre acessos locais e remotos.

Os resultados demonstram assim que o desempenho dos algoritmos é mais afetado ao nível da largura de banda da memória do que com a perda de desempenho implicada com as travessias do conetor de interligação dos processadores. Ou seja, é mais importante maximizar a largura de banda usada do que maximizar a percentagem de acessos à memória local.

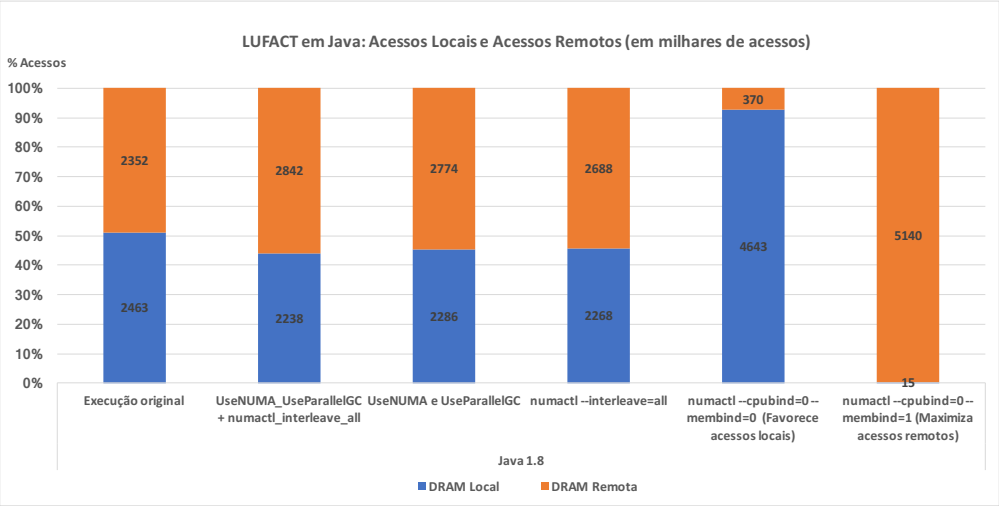


Figura 23.: LUFACT Java: Acessos Locais e Remotos (tamanho em DRAM - 16000x16000).

4.7 TESTES DE AFINIDADES COM IMPLEMENTAÇÕES EM C

Apesar do controlo das alocações em secções anteriores ter verificado ganhos de desempenho, é importante perceber se estes ganhos se aproximam dos ganhos de desempenho que são possíveis obter com este controlo noutras linguagens de programação.

Na secção 3.3 foram apresentadas abordagens de exploração de afinidades de fios de execução e abordagens de controlo de alocações na linguagem C. Nesta secção pretende-se aferir se os mecanismos de afinidade disponíveis para a linguagem *C*, asseguram um melhor desempenho nas arquiteturas NUMA comparativamente com o desempenho obtido em Java. Para isso foram explorados um conjunto de testes de afinidade para os mesmos algoritmos dos dois casos de estudo mas com as versões de memória partilhada implementadas na linguagem C.

Os testes de afinidade na linguagem C, incluem testes com variáveis de afinidade dos compiladores GCC (versão 5.3.0), o uso da ferramenta *numactl* com a mesma política de distribuição intercalada de páginas de memória realizada anteriormente nos testes em Java, e para a implementação do SOR foram ainda exploradas a diretiva *procbind* do OpenMP e as variáveis KMP_AFFINITY do ICC 16. Relativamente às variáveis de afinidade, foram explorados dois testes com a variável GOMP_CPU_AFFINITY

do GCC e KMP_AFFINITY do ICC 16 no algoritmo SOR. Os testes com a variável *GOMP_CPU_AFFINITY* realizam respetivamente, uma distribuição intercalada dos fios de execução pelos núcleos dos dois processadores e o segundo teste preenche primeiro os núcleos do primeiro processador. Os testes com esta variável foram feitos com afinidades explícitas em que cada fio de execução é mapeado num dado núcleo de um processador. As tabelas 2 e 3 ilustram o modo de distribuição dos fios de execução pelos núcleos em cada um dos casos. A imagem 15 pode ser utilizada para perceber o mapeamento dos fios de execução pelo núcleos do nó computacional usado nos testes.

# Fios de execução	**Processador**	**Núcleo**
1	0	P#0
2	0, 1	P#0 , P#8
4	0,1,0,1	P#0, P#8, P#1, P#9
8	0,1,0,1,0,1,0,1	P#0, P#8, P#1, P#9, P#2, P#10, P#3, P#11
(.....)		

Tabela 2.: Mapeamento de fios de execução realizados no teste *GOMP_CPU_AFFINITY Interlieving*.

# Fios de execução	**Processador**	**Núcleo**
1	0	P#0
2	0, 0	P#0, P#1
4	0,0,0,0	P#0, P#1, P#2, P#3, P#4
8	0,0,0,0,0,0,0,0	P#0, P#1, P#2, P#3, P#4 ... P#7
16	0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0	P#0, P#1, P#2, P#3, P#4 ... P#7 P#16 ... P#23
32	0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 1,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1	P#0, P#1, P#2, P#3, P#4 ... P#7 P#16 ... P#23 P#8, P#9 ... P#15 P#24, P#25 ... P#31

Tabela 3.: Mapeamento de fios de execução realizados no segundo teste com GOMP_CPU_AFFINITY (preenche primeiro o processador 0).

Note-se que o esquema de distribuição intercalada com o GOMP_CPU_AFFINITY apenas garante a distribuição dos fios de execução. Deste modo, não existe controlo sobre as alocações de memória. Os testes com a variável KMP_AFFINITY utilizam

as políticas de afinidade *scatter* e *compact* já pré-programadas à semelhança dos testes realizados com a diretiva *procbind*.

4.8 ANÁLISE DO IMPACTO DE AFINIDADES COM IMPLEMENTAÇÕES EM C

Nas figuras 24 e 25 são apresentados os resultados dos testes realizados com afinidades em C. Numa análise inicial percebe-se que as versões anteriores em Java de ambos os algoritmos escalavam melhor com o uso da ferramenta *numactl*.

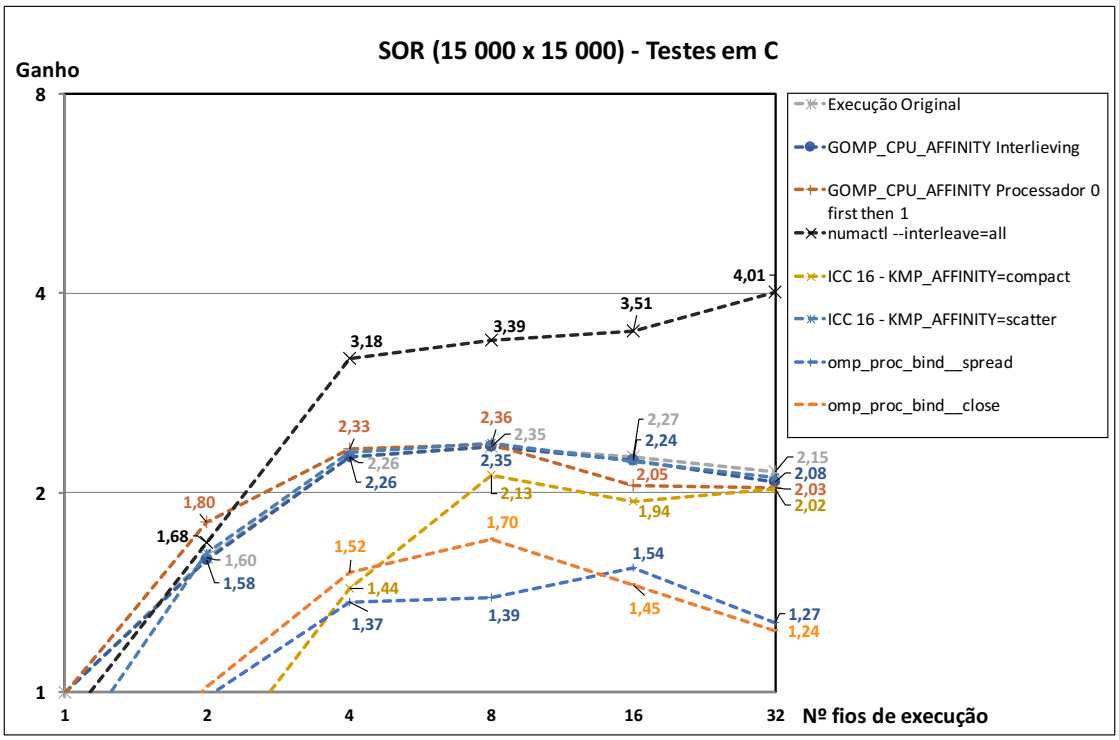


Figura 24.: SOR C: ganhos desempenho com afinidades para dados em DRAM (15000x15000).

Para os testes em C também se verifica uma tendência semelhante aos resultados anteriores em Java: quando o algoritmo é executado com um número de fios de execução relativamente pequeno (até 2 ou 4 fios de execução), é preferivel ter os fios de execução mapeados no processador 0. A partir desse limite, ou se distribuem os fios de execução e a memória intercaladamente pelos nós, ou a tendência quando o número de fios de execução aumenta, é o algoritmo deixar de escalar. Note-se no entanto que isto não acontece com o teste *numactl –interleave=all*. Após o limite dos 4 fios de execução, o algoritmo continua a escalar com o aumento do número de threads não existindo in-

versão da curva. Esta propriedade verificou-se nos testes em C e Java para dados na DRAM. A diferença, é que na generalidade, as versões Java com afinidades nos dois algoritmos, apresentam ganhos maiores. Os mapeamentos intercalados de fios de execução com a variável GOMP_CPU_AFFINITY (GCC) possuem basicamente o mesmo desempenho que o teste equivalente com a variável KMP_AFFINITY da Intel no entanto as afinidades com uso do KMP_AFFINITY e com as diferentes estratégias com OpenMP revelaram-se ineficientes uma vez que as suas curvas estão abaixo da curva da execução original.

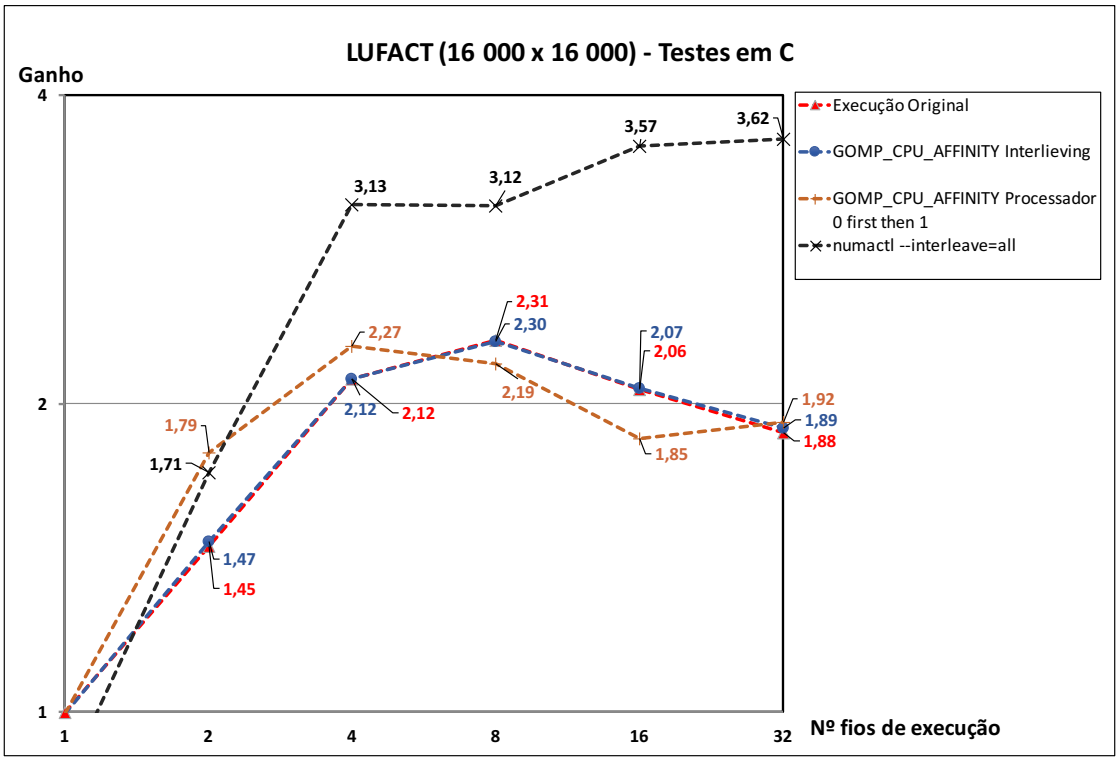


Figura 25.: LUFACT C: ganhos desempenho com afinidades para tamanho em DRAM (16000x16000)

4.9 ANÁLISE DA LARGURA DE BANDA E DOS ACESSOS DAS VERSÕES EM C

Á semelhança do que acontece para as implementações em Java, também nas implementações em C, as execuções originais são realizadas utilizando a memória de apenas

um dos processadores. As figuras 26 e 27, revelam que essa assimetria na largura de banda é ainda maior à que se verificou para as implementações em Java.

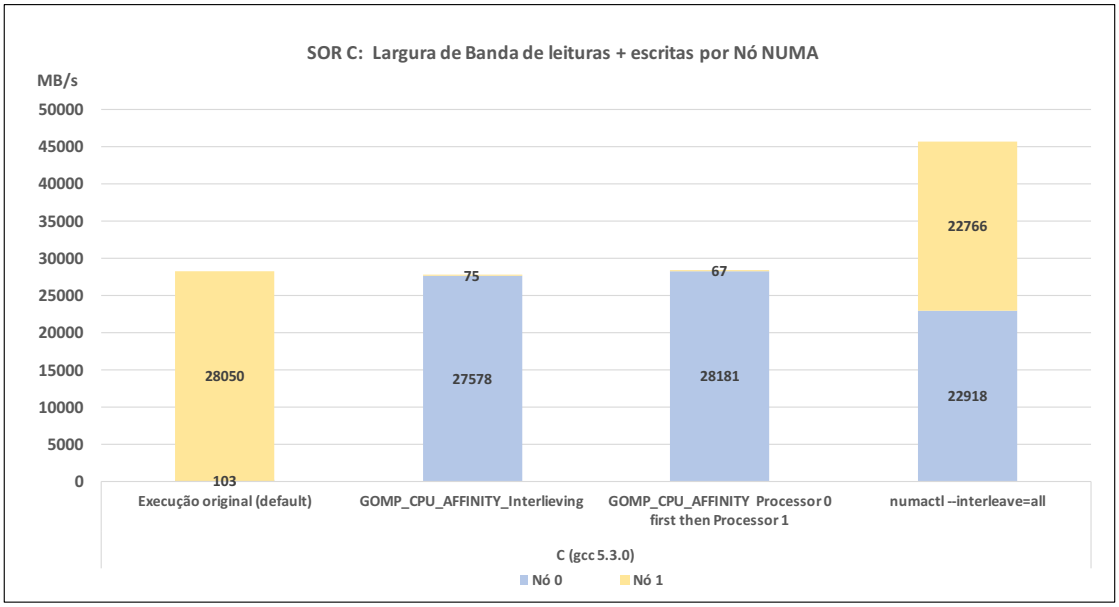


Figura 26.: SOR C: Largura de banda para o tamanho em DRAM (15000x15000).

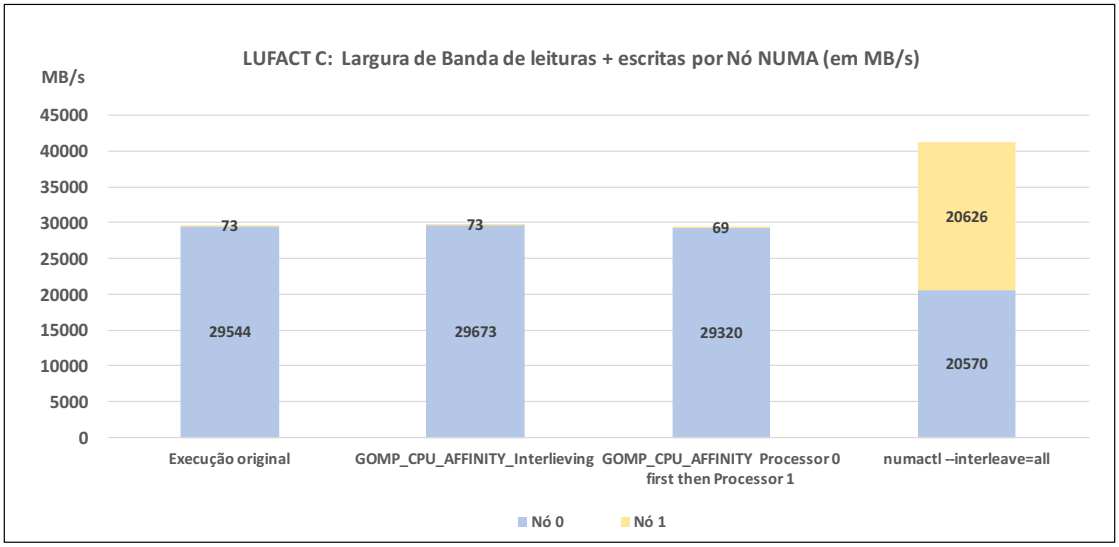


Figura 27.: LUFACT C: Largura de banda para o tamanho em DRAM (16000x16000).

Os testes com a variável GOMP_CPU_AFFINITY não revelaram ganhos significativos à excepção do teste que preenche o primeiro processador até dois fios de execução.

		Largura de banda máxima total (MB/s)	Ganho largura de banda	Ganho em tempo de execução (Melhores testes)
SOR (Java)	Original	24 471	**1.44**	**1.8**
	Melhor afinidade	35 303		
SOR (C)	Original	28 153	1,62	1,71
	Melhor afinidade	45 684		
LUFACT (Java)	Original	28 812	1,36	1,56
	Melhor afinidade	39 142		
LUFACT (C)	Original	29 617	**1,39**	**1,57**
	Melhor afinidade	41195		

Tabela 4.: Quadro resumo com os melhores resultados de largura de banda e tempo de execução.

Em termos de largura de banda verifica-se que apesar da realização do mapeamento de forma intercalada dos fios de execução (8 fios de execução neste teste) pelos processadores, a memória utilizada é a memória do processador 0. Por forma a otimizar a largura de banda, os fios de execução a correr no processador 1 deveriam utilizar a DRAM do processador 1. No entanto, esta variável não permite controlar essas alocações de memória em função do mapeamento escolhido para os fios de execução. A realização do teste *numactl –interleave=all* ao equilibrar as alocações de memória pelos dois nós, contribuiu para um ganho de 1.62 vezes em largura de banda no SOR, e 1.39 no LUFACT. Estes ganhos em largura de banda traduzem-se em ganhos em tempo de execução de 1.71 no SOR e cerca de 1.57 no LUFACT considerando o melhor ganho do *numactl* com o melhor ganho obtido com a execução original. É de notar que este teste com *numactl* registou o maior valor de largura de banda de todos os testes realizados nas implementações em C e Java.

Até ao momento é possível resumir os resultados obtidos para as versões em Java e C de ambos os algoritmos para os dados em DRAM de acordo com a tabela 4.

Os resultados dos contadores revelam que à semelhança das implementações em Java também nas versões em C o número de acessos locais e remotos é o mesmo a menos que seja forçada uma política que maximize os acessos locais como acontece com o teste da variável GOMP_CPU_AFFINITY. Uma vez que o resultado para o LUFACT é essencialmente o mesmo, o gráfico correspondente encontra-se no anexo B.

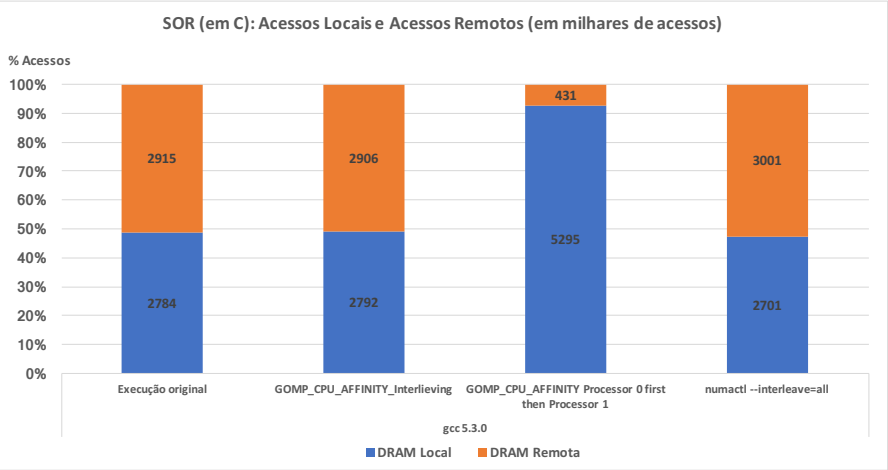


Figura 28.: SOR C: Acessos Locais e Remotos (tamanho em DRAM - 15000x15000).

4.10 IMPLEMENTAÇÕES FIRST TOUCH EM C

A técnica *First Touch* foi já apresentada na secção 3.5 e referida como uma técnica que permite diminuir a distância a que fios de execução se encontram dos dados de memória. A memória de cada fio de execução é alocada no mesmo nó do processador em que o fio de execução se encontra. Isto é garantido quando o fio de execução realiza o *toque* numa região de memória (por exemplo, através da inicialização de uma posição, ou conjunto de posições de um vetor). A aplicação desta técnica, ao contrário das técnicas abordadas até agora, implica a alteração do código fonte.

A implementação da técnica pode ser realizada quando uma região de código segue um padrão em que é possível identificar as seguintes fases:

1. Alocação de uma estrutura de dados $\boldsymbol{E}$;
2. Inicialização de $\boldsymbol{E}$;
3. Realização de computação em paralelo com $\boldsymbol{E}$;

Sobre uma zona de código com este padrão, a aplicação da técnica resume-se à junção dos dois últimos passos dentro da mesma região paralela. O objetivo desta junção é garantir que o fio de execução que inicializa uma região de memória, fica com as páginas

de memória associada a essa região na memória do processador onde o fio de execução está a executar. Foram realizadas três implementações das versões *First Touch* a partir do código fonte utilizado desde o início dos testes: duas implementações diferentes para o algoritmo SOR e outra para o algoritmo LUFACT.

As duas implementações *First Touch* para o algoritmo SOR baseiam-se em duas abordagens diferentes para realização do primeiro passo referente à alocação da estrutura de dados. A primeira utiliza a versão original do SOR em C utilizado neste estudo invocando apenas uma vez a função *malloc*. A segunda utiliza uma versão de vetor de apontadores para representar a matriz envolvida no algoritmo e realizar as alocações recorrendo a múltiplas chamadas à função *malloc*.

A implementação *First Touch* do LUFACT é baseada na versão C original do algoritmo sem modicação das estruturas de dados. Nas sub secções seguintes é explorada cada uma destas implementações e realizado um teste de desempenho por forma a perceber o impacto desta técnica no desempenho e poder comparar esta abordagem com as abordagens de afinidade realizadas anteriormente que não implicaram alterações ao nível do código.

4.10.1 *Implementação First Touch do SOR com uma alocação*

A implementação da versão *First Touch* com uma alocação utiliza essencialmente a mesma versão do código original. A aplicação da técnica é realizada na função com o *kernel* do algoritmo. Na seguinte figura apresenta-se a implementação original.

```
1  /* Alocacao da matriz G */
2  double (*G) [sor->N] = malloc(sizeof *G * sor->M);
3
4  /* Inicializacao matriz G em RandomMatriz */
5  RandomMatrix(sor->M, sor->N, G);
6
7  /* Computacao com matriz G em paralelo */
8  #pragma omp parallel
9  {
10     sor_simulation (1.25, sor->M, sor->N, G, sor->JACOBI_NUM_ITER,
11                     total_threads, sync);
12 }
```

Como a matriz G utilizada no sistema de equações do algoritmo é inicializada na função *RandomMatrix*, a versão *First Touch* implica uma mudança da função. Com esta mudança passa a ser possível juntar na mesma região paralela a fase de inicialização e de computação. A fase de computação é realiza pela função *sor_simulation* que recebe a matriz G como argumento. Na próxima figura apresenta-se a aplicação da técnica *First Touch*.

```
1  /* Alocacao da matriz G */
2  double (*G) [sor->N] = malloc(sizeof *G * sor->M);
3
4  #pragma omp parallel
5  {
6     /* Inicializacao */
7     #pragma omp for nowait
8     for (int i=0; i<sor->M; i++){
9        for (int j=0; j<sor->N; j++){
10          G[i][j] = (((double)rand()/(double)RAND_MAX))* 1e-6;
11       }
12    }
13    (...)
14    #pragma omp barrier
15
16    /* Computacao com matriz G em paralelo */
17    sor_simulation(1.25, sor->M, sor->N, G, sor->JACOBI_NUM_ITER,
          total_threads, sync);
18 }
```

Nesta abordagem, a fase de alocação não foi alterada sendo realizada pelo fio de execução mestre, uma única vez. Na próxima implementação First Touch será usada uma abordagem em que a matriz está representada segundo um vetor de apontadores, onde a fase de alocação pode ser realizada também em paralelo.

A nível de impacto em tempo de execução, é possível verificar na figura 29 que a execução sem afinidades escala melhor que a implementação sem *First Touch*. Á semelhança da implementação original, esta implementação escala também até aos 8 fios de execução. Ao analisar o gráfico constatamos que o esforço adicional de programação aumenta o desempenho do algoritmo a um nível que não se consegue atingir com as outras abordagens.

Com esta implementação *First Touch* o algoritmo apresentou uma ganho até 5 vezes. No entanto, comparativamente à versão em C original, o desempenho dos restantes de testes de afinidade, aumentou à excepção do teste com *numactl* que foi mais eficiente que a versão C orignal apenas para 32 fios de execução. No entanto, de desempenho dos testes de afinidade não foi suficiente para ficar acima da curva a cinzento correspondente à execução da implementação *First Touch*, sem afinidades.

Conclui-se que o esforço de programação adicional na codificação da versão *First Touch* é compensado porque além de fazer com que o algoritmo escale mais, torna-o mais independente de outras ferramentas.

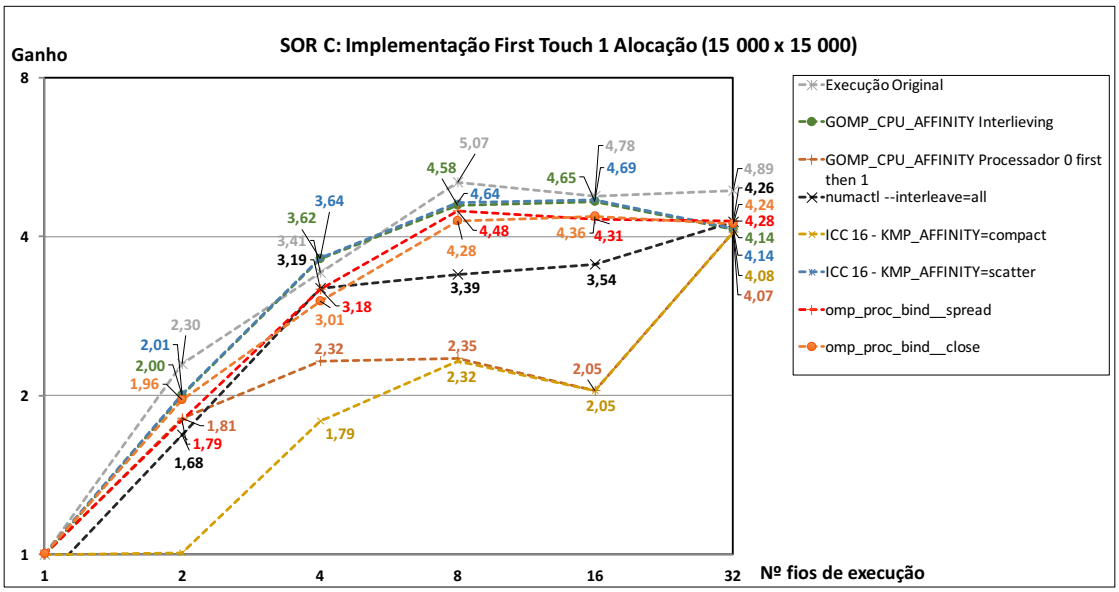


Figura 29.: SOR First Touch (1 Alocação): ganhos de desempenho (tamanho em DRAM - 15000x15000).

A análise da largura de banda na figura 31, demonstra que nesta implementação a execução original distribui melhor a largura de banda de memória, tal como a ferramenta *numactl* fez nos testes anteriores, sugerindo mais uma vez que os ganhos de desempenho derivam da distribuíção da memória alocada pelos nós.

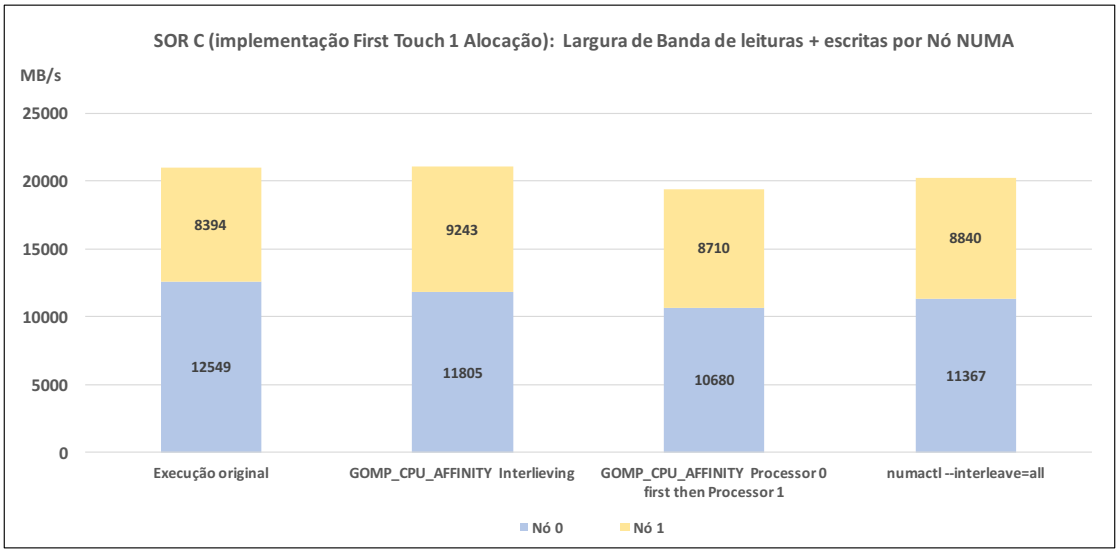


Figura 30.: SOR First Touch (1 Alocação): Largura de banda (tamanho em DRAM - 15000x15000).

Ao contrário do que se vinha a verificar para os testes anteriores que registavam um número de acessos remotos superior, para esta implementação First Touch existe a situação contrária. Todos os testes apresentam um número de acessos locais superior sugerindo que além da implementação já distribuir melhor a largura de banda pelos dois nós NUMA, também aumenta a quantidade de acessos locais. Este já seria um resultado esperado uma vez que devido à forma como as alocações de memória são realizadas nos nós, os fios de execução já possuem os dados que irão aceder no correspondente nó local não sendo verificada a necessidade de realização de acessos remotos.

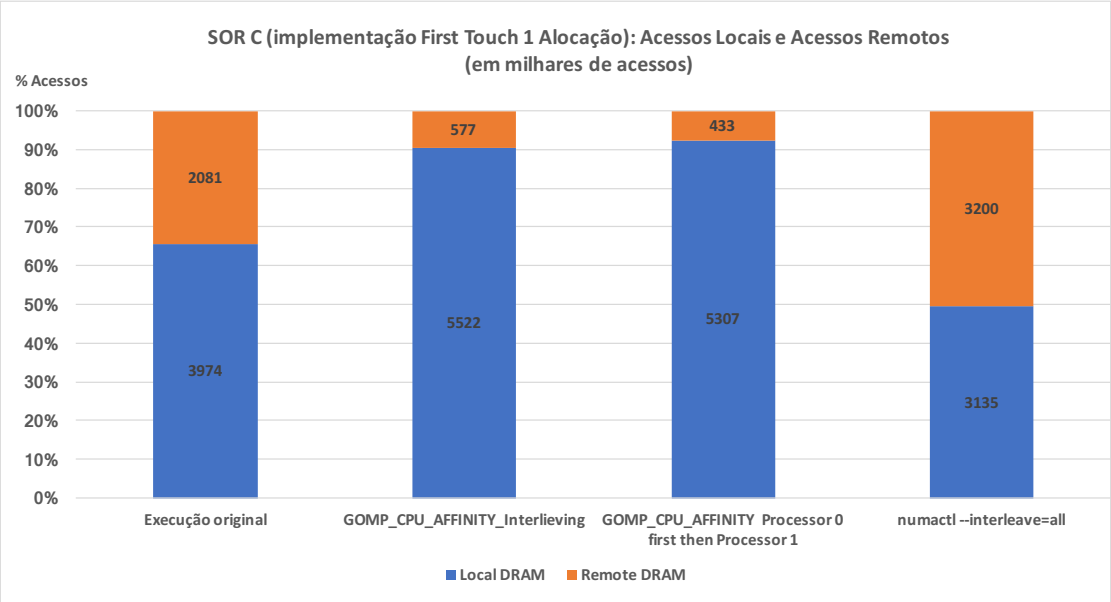


Figura 31.: SOR First Touch (1 Alocação): Acessos Locais e Remotos (tamanho em DRAM - 15000x15000).

4.10.2 *Implementação First Touch alternativa (SOR) com múltiplas alocações*

A implementação First Touch anterior foi aplicada à implementação original do algoritmo em C. Esta implementação *First Touch* alternativa aplica o mesmo princípio da anterior mas utiliza uma versão adaptada do código fonte por forma a representar a matriz como um vetor de apontadores, e realizar a alocação sequencialmente por linhas. De acordo com o gráfico de ganho 32 é possível concluir que esta versão apresenta um desempenho inferior em comparação com a implementação com uma alocação. Os restantes testes de afinidade para esta implementação, permanecem bastante semelhantes aos obtidos com a versão *First Touch* para uma alocação.

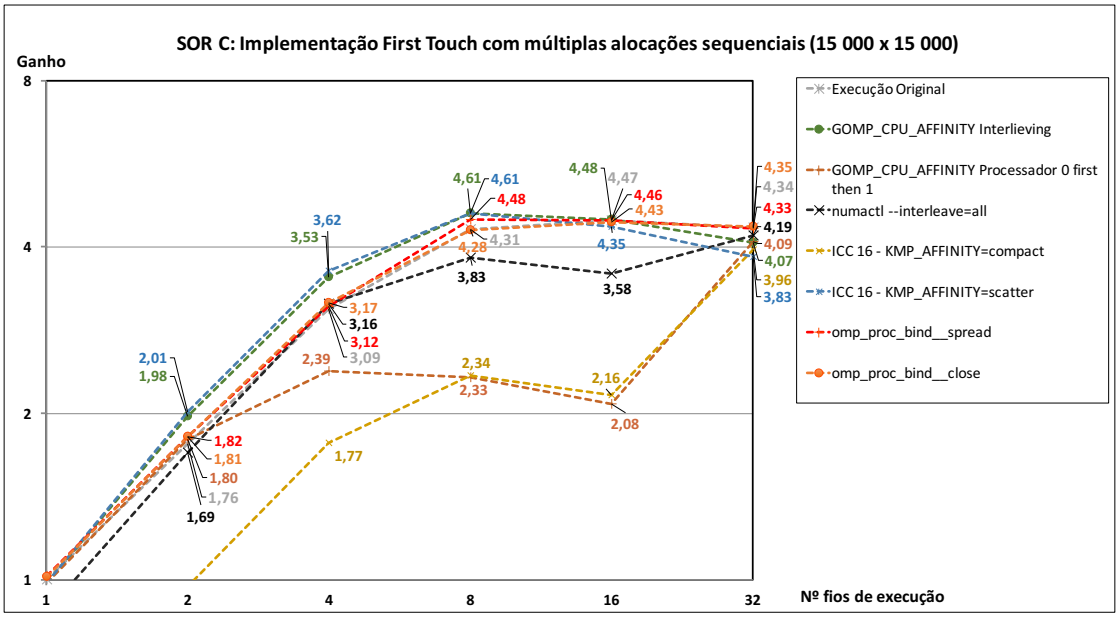


Figura 32.: SOR First Touch (Múltiplas alocações sequenciais): ganhos de desempenho (tamanho em DRAM - 15000x15000).

A figura 33 permite concluir que a alteração da estrutura de dados e as alocações realizadas por linhas não trazem melhorias de desempenho em termos de largura de banda. Á semelhança da versão *First Touch* anterior, o número de acessos locais remotos também verifica um predomínio do número de acessos locais sobre o número de acessos remotos. Dada a semelhança dos gráficos destes acessos, o gráfico correspondente a esta versão encontra-se no anexo B.

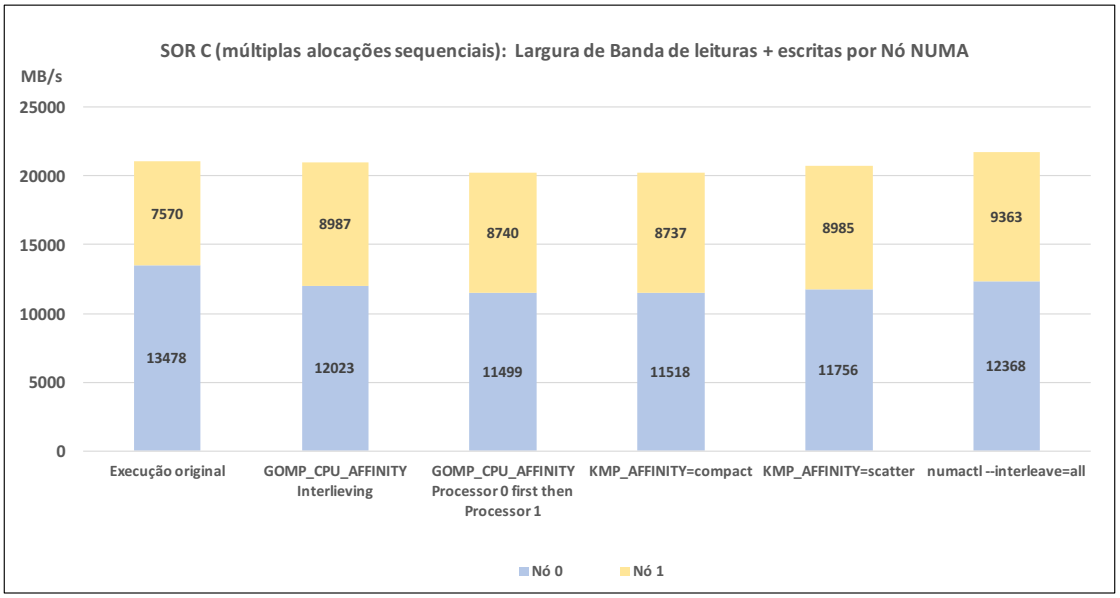


Figura 33.: SOR First Touch (Múltiplas alocações sequenciais): Largura de banda (tamanho em DRAM - 15000x15000).

4.10.3 *Implementação First Touch do LUFACT*

A nível de implementação a versão *First Touch* do LUFACT assemelha-se às versões anteriores. A única diferença é que nesta implementação, a inicialização em vez de ser realizada no mesmo corpo da função, é chamada uma função auxiliar de inicialização onde a técnica é aplicada de igual forma. A figura 34 demonstra que à semelhança da versão First Touch do SOR, esta versão reflete uma melhoria de desempenho. Nesta versão, atinge-se um ganho máximo de 4.36 vezes comparativamente à execução original sem *First Touch*. Também para o LUFACT, os restantes testes de afinidade mantêm um comportamento semelhante a nível de ganho em tempo de execução à exceção do teste com GOMP_CPU_AFFINITY que verificou um desempenho superior até aos 4 fios de execução. O teste com *numactl* com a política de alocação intercalada não revelou ter impacto nestas implementações *First Touch*.

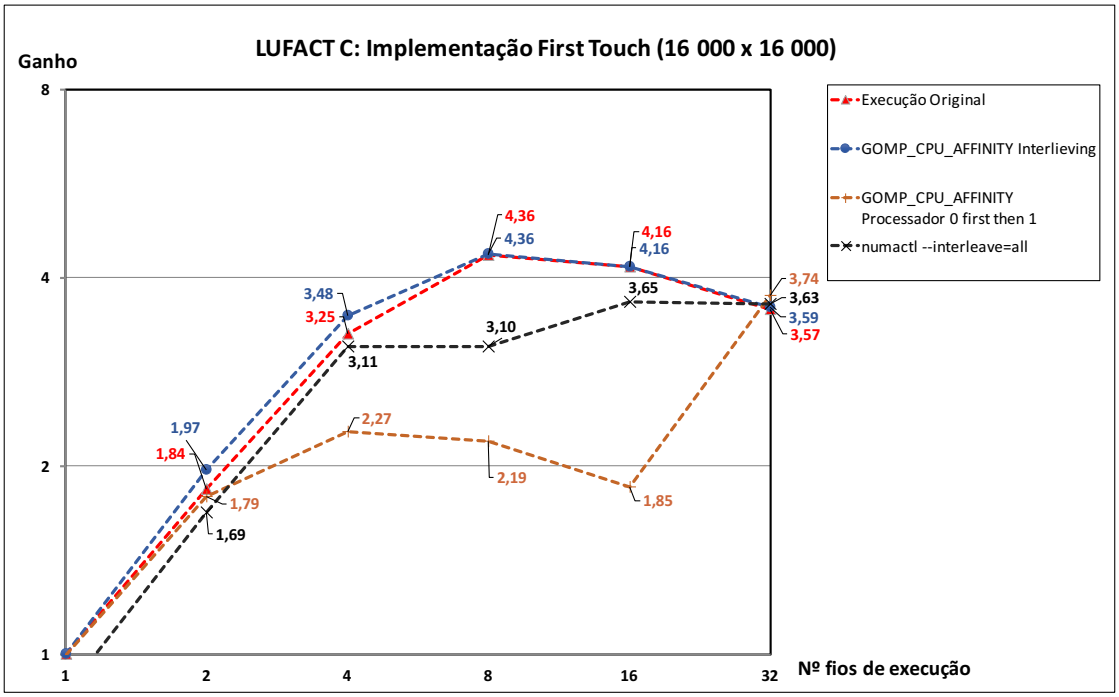


Figura 34.: LUFACT First Touch: ganhos de desempenho (tamanho em DRAM - 16000x16000).

Os gráficos das figuras 35 revelam que praticamente todos os testes beneficiaram diretamente dessa aplicação. As larguras de banda aumentaram pelo menos 1.5 vezes face à execução original em C sem *First Touch*. Enquanto que a versão sem *First Touch* do algoritmo registou praticamente o mesmo número de acessos locais e remotos, esta implementação aumentou para cerca de 90% o número de acessos locais.

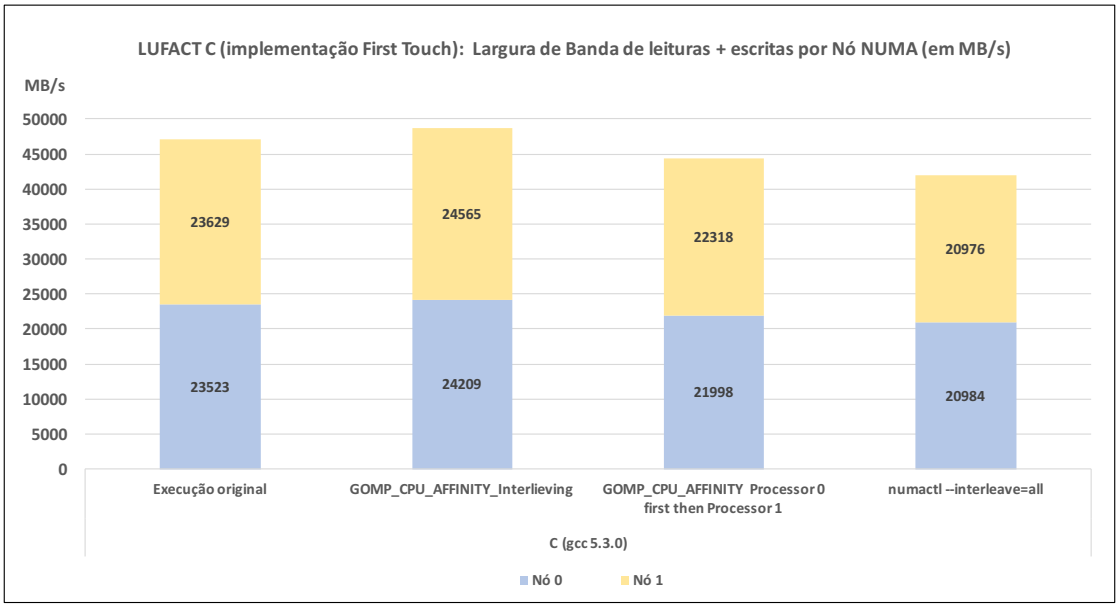


Figura 35.: LUFACT First Touch:largura de banda (tamanho em DRAM - 16000x16000).

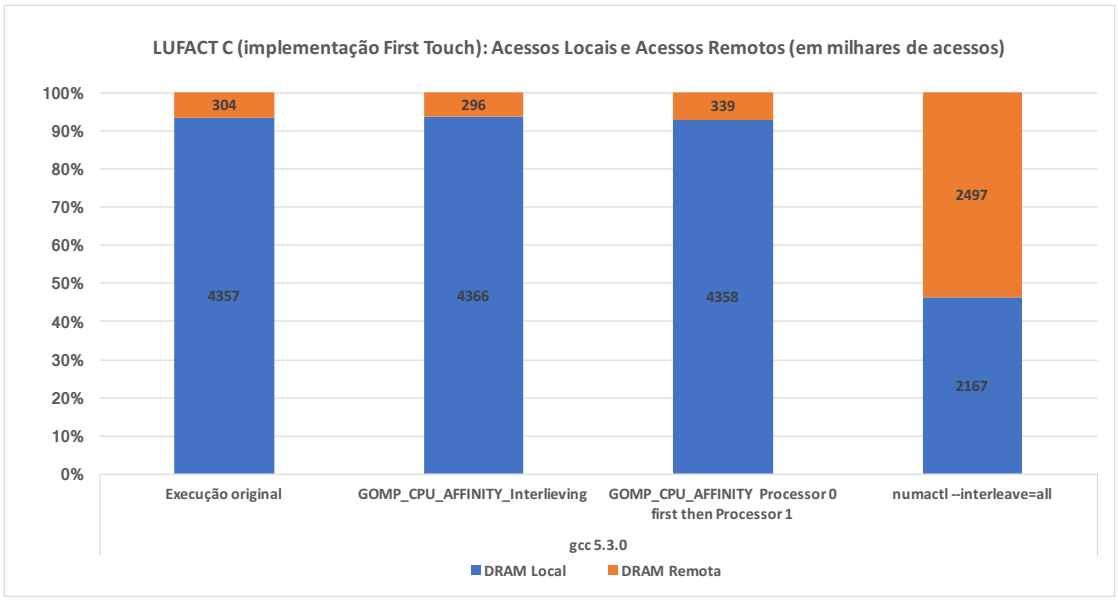


Figura 36.: LUFACT First Touch: Acessos Locais e Remotos (tamanho em DRAM - 16000x16000).

4.11 CONSIDERAÇÕES FINAIS: RESUMO DOS MELHORES RESULTADOS

Ao longo do estudo foram apresentadas diversas abordagens de afinidade com diferentes impactos no desempenho. A tabela 5 resume os resultados dos melhores ganhos de desempenho alcançados para as diferentes versões em C e Java e as suas versões melhoradas.

	Melhores ganhos em tempo de execução		
	Original	**Com afinidades**	**Ganho adicional**
SOR Java	2.44	4.40	1.8
LUFACT Java	2.71	4.22	1.56
SOR C	2.35	4.01	1.71
SOR C c/ First Touch (1 Alocação)	2.35 (SOR C)	5.07	**2.16**
SOR C c/First Touch (Múlt.Aloc.)	2.35 (SOR C)	4.47	**1.90**
LUFACT C	2.31	3.62	1.56
LUFACT C c/ First Touch	2.31	4.36	**1.89**

Tabela 5.: Melhores ganhos conseguidos com afinidades (para testes com dados em DRAM).

É possível verificar que as implementações *First Touch* na linguagem C superam os ganhos que se conseguem obter com as afinidades em Java ou com as diferentes otimizações disponíveis na JVM para arquiteturas NUMA. O uso de estratégias de afinidade contribuíram para aumentar os ganhos nas aplicações Java, no entanto estes resultados demonstram que esses ganhos são superados pelas versões First Touch implementadas em C.

5

CONCLUSÕES E DESAFIOS PARA TRABALHO FUTURO

Nesta tese foi feito um estudo sobre o impacto das arquiteturas NUMA no desempenho das aplicações Java. Numa primeira fase foi feito um estudo sobre a organização e desenho das arquiteturas NUMA. Este estudo revelou que a organização de memória destas arquiteturas permite atenuar limitações de escalabilidade verificadas nas arquiteturas SMP. No entanto, com a nova organização de memória, surgem novos desafios relacionados com o aproveitamento eficiente da largura de banda adicional disponível nos acessos à memória.

A interação dos sistemas operativos mais recentes com este tipo de arquiteturas revelam que a afinidade de fios de execução e a distribuição das páginas pelos bancos de memória principal, podem ter impacto no desempenho das aplicações nas arquitetura NUMA. Com este estudo ficou demonstrado que o controlo sobre a afinidade de fios de execução e sobre a distribuição de páginas de memória, deve ser tido em conta por forma a aumentar o desempenho das aplicações Java (e *C*). Essa manipulação pode ser feita para aplicações em Java com ferramentas externas como *numactl*. Existem otimizações para arquiteturas NUMA que podem ser ativadas na JVM sendo que apenas a opção *-XX:+UseNUMA* verificou um impacto no desempenho significativo para os casos de estudo.

Com o decorrer do estudo verificou-se que o nível de suporte para exploração de afinidades em Java é, no entanto mais reduzido comparativamente a linguagens mais antigas como a linguagem C. Em Java a exploração de afinidades está limitada ao uso de ferramentas e/ou bibliotecas de afinidade externas enquanto que para a linguagem C existem mais alternativas: APIs de paralelismo, chamadas a sistema, variáveis de afinidade dos compiladores, e técnicas ao nível da programação (*First Touch*).

Com este estudo verificou-se que os algoritmos *Successive Over Relaxation* e fatorização LU em Java são beneficiados a nível de desempenho com o controlo da afinidade dos fios de execução e das alocações de memória numa arquitetura NUMA *2-socket*. A ativação do alocador *NUMA aware* através da opção *-XX:+UseNUMA* revelou ter pouco impacto para matrizes em cache L3, mas esse impacto é significativo para tamanhos de dados em DRAM em ambos os algoritmos.

A ferramenta *numactl* apesar de se tratar de uma ferramenta externa conseguiu ganhos de desempenho superiores aos ganhos obtidos com as optimizações fornecidas pela própria JVM em tamanhos de dados em DRAM. Este estudo revela que das diferentes configurações de afinidades estudadas, os maiores ganhos de desempenho para as duas aplicações Java são conseguidos intercalando as páginas de memória pelos dois nós com a ferramenta *numactl*. Os testes realizados com a ferramenta PCM, evidenciaram uma relação direta entre o ganho e a largura de banda de acesso à DRAM utilizada. Para os testes realizados, quanto maior a largura de banda, maior o ganho conseguido.

Os testes à largura de banda na execução original dos dois algoritmos em Java, mostram que a tendência da JVM é de executar as aplicações fazendo uso de apenas um dos bancos de memória. Os testes mostram igualmente que essa utilização é, no entanto, ineficiente nas arquiteturas NUMA. Mapear os fios de execução em apenas um processador e utilizar apenas a sua memória local minimiza o número de acessos remotos. No entanto os testes realizados nesta tese, verificaram que é preferível distribuír as páginas pelos diferentes bancos de memória. Apesar de poder existir um aumento do número de acessos remotos, a largura de banda utilizada é superior e encontra-se melhor distribuída pelos dois bancos de memória principal.

Nas implementações dos mesmos algoritmos em linguagem C, verificou-se que a alocação das páginas de memória seguindo uma política de alocação intercalada obteve igualmente os melhores resultados quer a nível de ganho em tempo de execução, quer a nível de largura de banda. As implementações em C (que não aplicam a técnica First Touch), à semelhança das implementações em Java, verificam uma assimetria de largura de banda nas execuções sem controlo de afinidades nos dois bancos de memória. O teste *numactl* com a opção *interleave* para estas implementações permitiu corrigir essa assimetria melhorando os ganhos em tempo de execução através do aumento da largura de banda total utilizada.

As implementações *First Touch* para ambos os algoritmos, permitiram diminuir os problemas de assimetria de largura de banda diretamente no código fonte. O desempenho com estas implementações registou o maior ganho de todos os testes em ambos os algoritmos. A implementação *First Touch* do algoritmo SOR registou um ganho adicional de 2.16 vezes face ao mesmo algoritmo em C original, e no caso do LUFACT a implementação *First Touch* registou um ganho adicional de 1.89 vezes.

A aplicação desta técnica de programação, em termos de alocações resultou numa distribuição mais eficiente das páginas de memória pelos diferentes nós NUMA. Por essa razão, a execução nestas implementações registou uma largura de banda mais simétrica logo à partida sem serem precisas quaisquer ferramentas externas ou outros mecanismos de controlo de afinidades adicionais. Os ganhos atingidos com as execuções das implementações *First Touch* assemelham-se aos ganhos que se obtiveram anteriormente com as implementações em C com uso da ferramenta *numactl* com *interleave*. Também se verificou a mesma semelhança de resultados em termos de largura de banda.

Ao longo do desenvolvimento da tese foi constatada a importância do controlo das afinidades dos fios de execução e distribuição da memória como forma principal de aumento de desempenho de aplicações em arquiteturas NUMA. No contexto deste estudo foram exploradas muitas ferramentas que exploram diferentes abordagens de afinidade de fios de execução e de dados de memória. Entre todas as ferramentas e testes realizados, verificou-se que apenas a técnica *First Touch*, a *flag -XX+UseNUMA* da JVM, e o *numactl* foram verdadeiramente úteis para aumentar o desempenho dos dois casos de estudo. As ferramentas do PCM, foram as ferramentas de *profilling* em relação à memória mais úteis de todas as que foram exploradas. Contudo, o suporte para um controlo flexível sobre as afinidades de fios de execução e alocações de memória em Java ainda é reduzido.

5.1 TRABALHO FUTURO

Sugere-se como trabalho futuro o desenvolvimento de uma biblioteca Java que permita controlar o mapeamento dos fios de execução da aplicação e alocações de memória nos nós. Dada a contínua aposta dos fabricantes em arquiteturas NUMA, o desenvolvimento de uma biblioteca a esse nível permitiria oferecer maior nível de suporte das

aplicações Java nas arquiteturas NUMA existentes, e um suporte mais flexível para arquiteturas NUMA futuras.

Os resultados experimentais desta tese foram centrados numa única arquitetura NUMA com 2 processadores. Foi feito um conjunto de testes em torno de diferentes configurações de afinidade, e adicionalmente testes de largura de banda e testes de contadores de memória. Não foi medido o impacto da alteração do tamanho das páginas de memória com aplicação das estratégias de afinidade. Uma vez que a ferramenta *numactl* oferece algum controlo sobre o tamanho das páginas, seria interessante num trabalho futuro, a reprodução dos testes Java realizados com variação do tamanho das páginas por forma a perceber o impacto dessa abordagem numa arquitetura NUMA. Assim como foi fixado o tamanho das páginas de memória, também foi fixada a arquitetura NUMA utilizada. Deste modo, sugere-se como trabalho futuro medir o impacto deste tipo de afinidades e da técnica *First-Touch* quando aplicada a uma arquitetura NUMA com um número de nós superior.

A biblioteca *CoralThreads* trata-se de uma biblioteca comercial para a qual foi feito um pedido de licença académica. Uma vez que esta licença não foi disponibilizada, optou-se por focar o estudo noutras abordagens de afinidade.

A biblioteca *Java Thread Library* esteve durante algum tempo sem ser mantida e não foi utilizada neste estudo. No entanto há um autor que extendeu esta biblioteca recentemente pelo que sugere-se a exploração da mesma a nível de trabalho futuro. No entanto, pelo estudo que foi feito sobre os mapeamentos utilizando as variáveis de afinidade e OpenMP para controlo manual dos fios de execução, as versões em C não verificaram ganhos de desempenho relevantes.

Ao longo da realização da tese houve a necessidade de explorar algumas ferramentas para tentar aceder aos contadores de hardware capazes de fornecer informação dos contadores de hardware sobre acessos locais, remotos, e transferências de dados entre cache e DRAM, ou entre processadores. Atualmente ainda existe alguma escassez de ferramentas que permitam, de uma forma precisa e facilitada, aceder a esse tipo de informação que possibilite um *profilling* facilitado para avaliar as questões de desempenho nas arquiteturas NUMA.

A

ANEXO A: TAMANHOS DE DADOS

SOR (Tamanho das Matrizes)	
Tipo: double (8 bytes)	
Dimensões	**Tamanho (em MB)**
1500x1500	17.16
15000x15000	1716.61

LUFACT (Tamanho das Matrizes)	
Tipo: double (8 bytes)	
Dimensões	**Tamanho (em MB)**
1000x1000	7.63
16000x16000	1953.1

Tabela 6.: Tamanho das matrizes para os algoritmos SOR and LUFACT.

B

ANEXO B: GRÁFICOS AUXILIARES

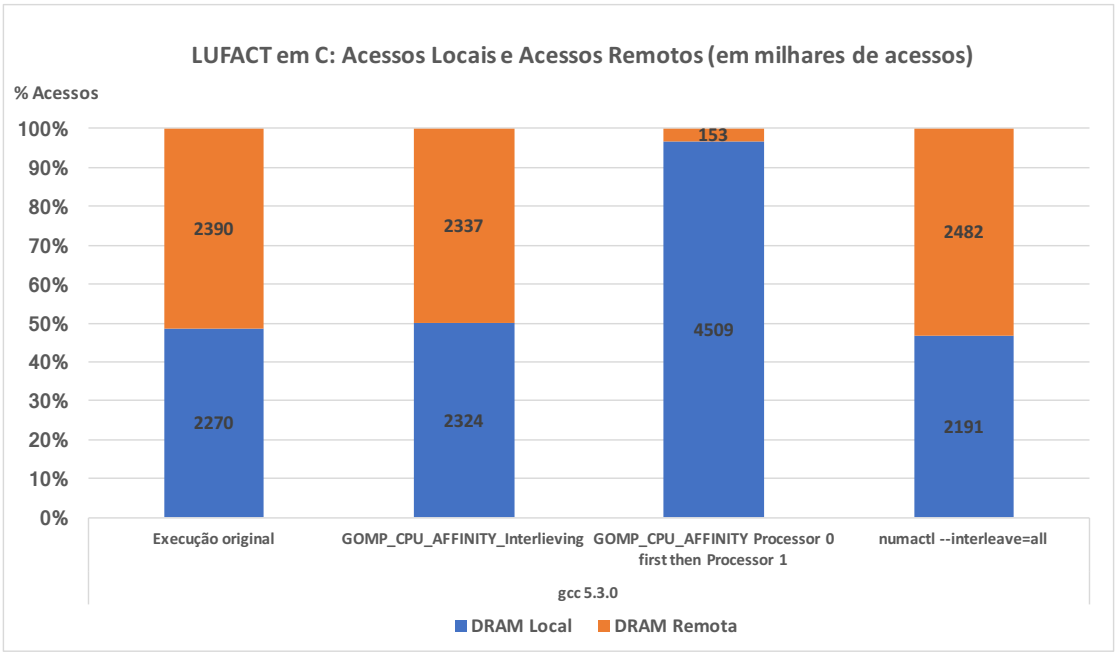


Figura 37.: LUFACT C: Acessos Locais e Remotos (tamanho em DRAM - 16000x16000).

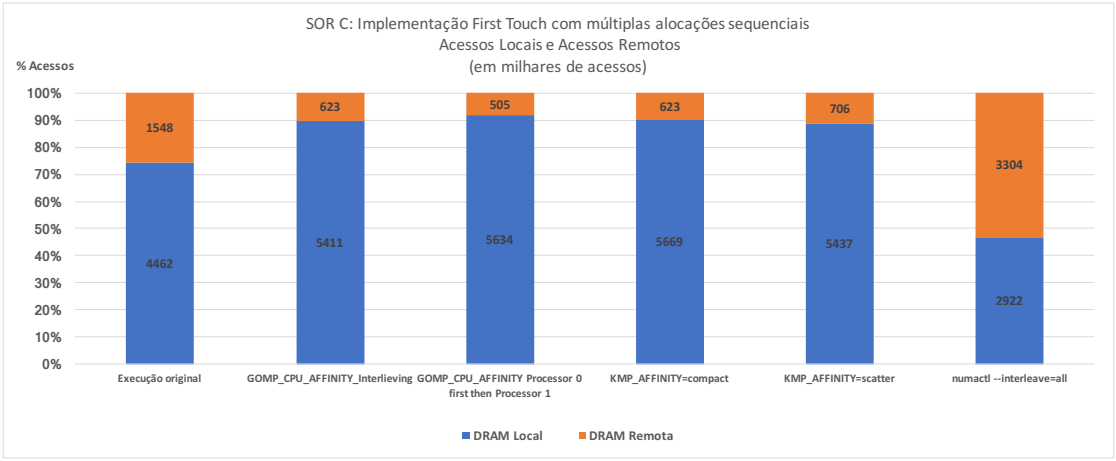


Figura 38.: SOR First Touch (Múltiplas alocações sequenciais): Acessos Locais e Remotos (tamanho em DRAM - 15000x15000).

BIBLIOGRAFIA

[1] Openhft: Java thread library. URL https://github.com/OpenHFT/Java-Thread-Affinity.

[2] NASA NAS High-End Computing Capability. Using intel openmp thread affinity for pinning. URL https://www.nas.nasa.gov/hecc/support/kb/using-intel-openmp-thread-affinity-for-pinning_285.html.

[3] IBM Knowledge Center. Environment variables for openmp. URL https://www.ibm.com/support/knowledgecenter/en/SSAT4T_15.1.4/com.ibm.xlf1514.lelinux.doc/proguide/ompplaces.html.

[4] Cache coherence (Wikipedia). URL https://en.wikipedia.org/wiki/Cache_coherence.

[5] CoralThreads. URL http://www.coralblocks.com/index.php/getting-started-with-coralthreads/.

[6] Intel Corporation. Product brief intel® xeon® processor e7-8800/4800/2800 v2 product families, . URL https://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-v2-family-brief.html.

[7] Intel Corporation. Intel® xeon® bronze 3106 processor specifications, . URL http://ark.intel.com/products/123540.

[8] Intel Corporation. Intel® xeon® silver 4114 processor specifications, . URL http://ark.intel.com/products/123550.

[9] Intel Corporation. Intel® xeon® processor scalable family technical overview, . URL https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[10] Intel Corporation. Intel xeon scalable platform - the future-forward platform foundation for agile digital services, . URL https://www.servethehome.com/intel-xeon-scalable-processor-family-platform-level-overview/.

[11] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. Affinity-based thread and data mapping in shared memory systems. *ACM Comput. Surv.*, 49(4):64:1–64:38, December 2016. ISSN 0360-0300. doi: 10.1145/3006385. URL http://doi.acm.org/10.1145/3006385.

[12] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005. ISBN 0471467405.

[13] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. URL http://dx.doi.org/10.1109/TC.1972.5009071.

[14] Aaron Birkland (Cornell Center for Advanced Computing) with contributions from: Texas Advanced Computing Center and Intel Corporation. Threading affinity. URL https://cvw.cac.cornell.edu/mic/affinity.

[15] Linux Support for NUMA Hardware FAQ. URL http://lse.sourceforge.net/numa/faq/.

[16] Processor Counter Monitor (PCM) Github. URL https://github.com/opcm/pcm.

[17] Lucio Grandinetti. *Advances in High-Performance Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997. ISBN 0792345509.

[18] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013. ISBN 0133390098, 9780133390094.

[19] VAX-System User Guide. Vax 8820/8830/8840 system hardware user's guide. URL http://bitsavers.trailing-edge.com/pdf/dec/vax/

8800/EK-8840H-UG-001_88xx_System_Hardware_Users_Guide_Mar88.pdf.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.

[21] Christoph Lameter. Numa (non-uniform memory access): An overview. *ACM Queue*, 11(7):40:40–40:51, July 2013. ISSN 1542-7730. doi: 10.1145/2508834.2513149. URL http://doi.acm.org/10.1145/2508834.2513149.

[22] Xu Liu and John Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555271. URL http://doi.acm.org/10.1145/2555243.2555271.

[23] Jake Edge LWN. Perfcounters added to the mainline. URL https://lwn.net/Articles/339361/.

[24] Kasim M. Al-Aubidy. Memory systems in pipelined processors (lecture 12) - philadelphia university. URL https://www.philadelphia.edu.jo/academics/kaubaidy/uploads/ACA-Lect12.pdf.

[25] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *Crossroads*, 5(3es), April 1999. ISSN 1528-4972. doi: 10.1145/357783.331677. URL http://doi.acm.org/10.1145/357783.331677.

[26] Migratepages Linux Administrators Manual. URL http://man7.org/linux/man-pages/man8/migratepages.8.html.

[27] G. Mauler and M. Beebe. *Clustering Windows Server: A Road Map for Enterprise Solutions*. Elsevier Science, 2001. ISBN 9780080488493. URL https://books.google.pt/books?id=P5kx6MWu2eIC.

[28] Bruno Medeiros and João L. Sobral. Aomplib: An aspect library for large-scale multi-core parallel programming. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5117-3. doi: 10.1109/ICPP.2013.36. URL http://dx.doi.org/10.1109/ICPP.2013.36.

[29] Microway. Performance characteristics of common transports and buses. URL https://www.microway.com/knowledge-center-articles/performance-characteristics-of-common-transports-buses/.

[30] Edmond Chow Georgia Tech College of Computing. A course on high performance scientific computing on intel processors. URL https://www.cc.gatech.edu/~echow/ipcc/hpc-course/HPC-numa.pdf.

[31] Oracle-Java-SE-Documentation. Java hotspot^{TM} virtual machine performance enhancements. URL http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

[32] Linux Man Pages. numa_maps: overview of non-uniform memory architecture. URL https://www.systutorials.com/docs/linux/man/5-numa_maps/.

[33] QDPMA. Intel nehalem based systems with qpi. URL http://www.qdpma.com/systemarchitecture/systemarchitecture_qpi.html.

[34] Martin Schulz. *Shared memory programming on NUMA-based clusters using a general and open hybrid hardware, software Approach*. PhD thesis, Technical University Munich, Germany, 2001. URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/schulz.pdf.

[35] A. Schwarzenberg-Czerny. On matrix factorization and efficient least squares solution. *Astronomy and Astrophysics Supplement*, pages 405–410, April 1995. URL http://adsabs.harvard.edu/full/1995A%26AS..110..405S.

[36] João L. Sobral, Carlos A. Cunha, and Miguel P. Monteiro. Aspect oriented pluggable support for parallel computing. In Michel Daydé, José M. L. M Palma, Álvaro L. G. A Coutinho, Esther Pacitti, and João Correia Lopes, editors, *High*

Performance Computing for Computational Science - VECPAR 2006: 7th International Conference, Rio de Janeiro, Brazil, June 10-13, 2006, Revised Selected and Invited Papers, pages 93–106, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71351-7. doi: 10.1007/978-3-540-71351-7_8. URL https://doi.org/10.1007/978-3-540-71351-7_8.

[37] William Stallings. *Data and Computer Communications*. Prentice Hall Press, Upper Saddle River, NJ, USA, 9th edition, 2010. ISBN 0131392050, 9780131392052.

[38] Java Grande Forum Benchmark Suite. Epcc. URL https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index.php.

[39] Andrew S. Tanenbaum. *Structured Computer Organization (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. ISBN 0131485210.

[40] Jie Tao, Martin Schulz, and Wolfgang Karl. Simulation as a tool for optimizing memory accesses on numa machines. *Perform. Eval.*, 60(1-4):31–50, 2005. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/j.peva.2004.10.003.

[41] Victor Eijkhout TACC UTexas. Parallel programming in mpi and openmp - 23 openmp topic 8: Affinity. URL http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html.

[42] Martin H. Weik. Ballistic research laboratories electronic scientific computer. URL http://ed-thelen.org/comp-hist/BRL64-b.html#BURROUGHS-D825.

[43] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, March 2001. ISSN 0163-5964. doi: 10.1145/373574.373576. URL http://doi.acm.org/10.1145/373574.373576.

[44] Gregory V. Wilson. The history of the development of parallel computing. URL http://ei.cs.vt.edu/~history/Parallel.html.

[45] René Müller Vijayshankar Raman Guy M. Lohman Yinan Li, Ippokratis Pandis. Numa-aware algorithms: the case of data shuffling. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, January 2013. URL https://www.microsoft.com/en-us/research/publication/numa-aware-algorithms-the-case-of-data-shuffling/.